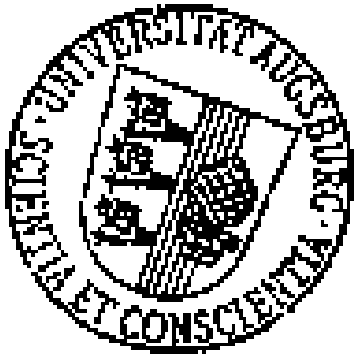
Dual Compilation for Hardware and Software

Ewald Frensch, Universität Augsburg



Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

Erstgutachter: Prof. Dr. Bernhard Möller
Zweitgutachter: Prof. Dr. Theo Ungerer
Prüfer: Prof. Dr. Rainer Lienhart
Prüfer: Prof. Dr. Bernhard Bauer

Abstract

For many years hardware designers and software programmers have been using different description languages to implement algorithms. For software systems the language C (and it's derivatives like C++) made its way, whereas hardware systems are implemented in one of the hardware description languages like VHDL, Verilog or Able.

The main reason why the hardware and software design methodologies have branched is the fact that hardware and software was developed by different persons. Hardware engineers are used to structural thinking whereas software engineers think algorithmically. This fact is reflected in the tools used for the development. In the last few years a new type of engineer has been established: the System Engineer. This was a result of the awareness that hardware and software can not be treated totally separately. They have influence on each other. The System Engineers have realized that for many reasons it is very advantageous to have one language for both, hardware and software (for rapid prototyping, hardware/software split systems etc.).

A lot of new research topics arose when the discussion about Dual Compilation for Hardware and Software started. The most important question when talking about Dual Compilation is: Is it possible to combine high-level compilers for silicon gate devices with proven software methodologies without being afflicted with a flawed synthesis process?

Some of the members in the community discussing this item do not take care of an unflawed synthesis primarily, but they try to figure out the optimal design language and the optimal level of abstraction for the design entry[1]. Of course it is advantageous to have a common language like C and the level as high as possible in order to make the source code highly readable and testable. This may be the easiest way to bridge the gap between hardware und software, but it runs the risk to raise the design level at a theoretical and impractical level for real targets. R. Leupers et al. have focused their research on this item. Refer to [25].

A second group of the research community try to develop tools (i.e. compilers) for automatic generation of a synthesisable representation of the algorithm. We realise two main streams: The first one provides tools to transfer the source code into the RTL[2] implemented in a common hardware description language (like VHDL or Verilog). This level of abstraction is accepted to be synthesisable anyway. The second main stream is focused on the direct compilation from high level source code to silicon net lists.

A lot of papers concerning the hardware compilation from a normal programming language have been published in the last few years (for example Handel C by Ian Page et al. [2]). Handel C has been successfully implemented by Celoxica Inc [3]. These tools use asynchronous timing. This is typical for software system. Cycle accurate timing is one of

[1]The circuit description with the highest level of abstraction

[2]Register Transfer Level

the most important features for hardware systems. The synchronisation is done by special signals between concurrent threads. We are going a different way. We're using a global clock for synchronization. Using this mechanism, it is possible to hide the timing analysis. The compiler may predict the timing behaviour, whereas the programmer using Handel C must take care of the timing and the thread synchronisation on his own. Using our compiler the programmer can focus his interest on the algorithm even if accurate timing is required.

For this purpose, we propose a C-based programming language. This language is not real C but very similar, in order to enable "normal" software programmers to develop digital circuits without having detailed knowledge in hardware or hardware description languages respectively.

The compiler proposed in this thesis provides all the typical hardware-items like timing analysis, net list generation etc.

Contents

List of Figures

1 Introduction

Are VHDL and Verilog past their prime, soon to be replaced by C-like design languages, Professor Ian Page[3] asked a couple of years ago. He said a change is at hand. Two main reasons led him to predict this change: rising costs to create state-of-the-art chips with hundreds of millions of logic gates and the problems of attracting and retaining the engineers who can handle such complex chips. The discussion was accompanied by a significant change in the semiconductor landscape: the disproportional rise of the FPGAs[4]. The FPGAs made the development process more software-like. The way from a rapid prototype to a final device was much easier and cost-efficient than the refinement loop of an ASIC[5] design. Since the introduction of FPGAs for prototyping as well as for plant operations the development effort was reduced to the pure programming using one of the particularly developed hardware description languages. Since the hardware development process became more and more software-like it was a logical conclusion to bridge the gap between hardware and software.

Many examples have shown that algorithms intended to be implemented in hardware at the beginning of a project were finally implemented at least partially in software and vice versa. The switch from hardware to software is usually not planned but done due to the experiences collected during the development phase. The contrary switch, from software to hardware, is taken for many reasons:

- Evaluation of the algorithmic complexity
- Rapid prototyping
- Demonstration purposes
- Simulation, Verification and Test of the algorithms

Anyway, if intended or not, the hardware-software switch (or partitioning) require comfortable tools for design and implementation. The research results in the past have shown that it is better to use software programming languages rather than hardware description languages. But, we will show later that pure programming languages are not useful for this purpose. Some extensions and restrictions may apply to the syntax and semantics.

1.1 Aims of the work

The aim of this work was to develop a smart methodology for Dual Compilation (Hardware und Software) using a derivate of the programming language C. It was motivated by Professor Ian Page's research. Prof. Page was involved in the Hardware Compilation Group at the Oxford University for many years and published a lot of results concerning hardware compilation and

[3] Oxford University
[4] Field-Programmable Gate Arrays
[5] Application Specific Integrated Circuit

hardware/software codesign. A commercial spin-out company[6] founded by Prof. Page et al. has shown that his results are mature enough to leave the academic world and make their way into commercial products. An extensive study of the publications of different research groups[7] have shown that some deficiencies may be removed in future.

This thesis focuses on a new methodology for the implementation of a hardware compiler for a C-like programming language that is able to generate timing accurate silicon gates. The work shows that even if some extensions and restrictions are made to ANSI-C[8] it is still possible to use this language as a normal programming language. A code generator for the ARM[9] microcontroller was implemented for this purpose.

The compiler is intended to achieve the maturation of a technology demonstrator. Although, it was not the intention to produce a ready-for-market highly optimizing compiler that produces optimal code, a big part of the exploration was focused on optimisation techniques. Optimisation is a very important part of the compilation process (especially for hardware compilation).

Finally it was explored, how the work may be embedded in related research activities. Although the current research activities are mainly focused on the problem to figure out the best design entry, it will be shown that the work can excellently be embedded in the related activities.

1.2 Structure of the document

Within the next chapter a programming language is proposed which is applicable for dual compilation for hardware and software. Since it is based on the well-known C we have focused our interests on the differences. At the end of that chapter some examples containing typical applications demonstrate the usage of our C derivative. These examples have been chosen since they are often implemented in software as well as in hardware dependent on the field of application.

Chapter 3 shows the possibility to implement a real compiler which translates the source code into the assembler code of a real microcontroller. Without loss of generality we can say that the source code may be compiled for the assembler code of any currently available microcontroller even if the compiler generates code for a particular target architecture. The consideration of a software target is done only for completeness. It is evident that a programming language based on C is suitable for this purpose. The hardware target has been examined more closely. The entire compilation process is described but we focused upon the timing, since this is one of the greatest difficulties when compiling source code for hardware targets. The explanations are accompanied by some examples.

[6] Celoxica Ltd, 66 Milton Park, Abingdon, Oxfordshire, OX14 4RX, United Kingdom

[7] e.g. R. Leupers et al., RWTH Aachen

[8] American National Standards Institute

[9] ARM architecture CPU design or one of its derivatives developed by ARM Ltd (originally called The Acorn RISC Machine)

Since the focus of our consideration is the timing behaviour and its steering, we have separated this discussion into chapter 4. The timing is evaluated at intermediate code level. Therefore a formal description of this level is defined in that chapter.

Some experiments and their results are given in chapter 5. The execution and the corresponding results are explained at different levels of abstraction.

In chapter 6 we discuss two related bodies of works. The methods and results are compared to the works of Ian Page [1] et al. from Oxford University. This group has implemented algorithms having a similar aim. A second group of researchers from the University of Aachen have focused their interests on an optimal design philosophy. We have evaluated how our work can be embedded in this research area.

An outlook to future research and open problems is given in chapter 7. We discuss two main items:

- How can the methodology be improved?
- Is it possible to use the methodology and the corresponding compiler for real applications?

1.3 Technical Basics

As we've already mentioned, the syntax of our programming language is very similar to C. C was developed for programming sequential machines (microprocessors). To make it suitable for hardware designers we have to remove some constructs. On the other hand we have to add some constructs for concurrency and for input/output.

Usually the compilation of a C program is done in several steps (i.e. scanner, lexical analysis etc.) If all these steps are processed only once, we call the compiler a single-pass compiler, otherwise it is called a multi-pass compiler. For our purposes it is suitable to implement a multi pass compiler, since at least the net list refinement is implemented as a separate loop.

As we will see later in this thesis, the first steps of the compiler are implemented similar to a common C compiler, since the syntax and the lexical interpretation is very close to the ANSI C proposed by Kernighan and Ritchie (refer to [11]). One of the most important differences in the terms of grammar is that our C is not necessarily context-independent, since we need to support concurrent statements. Context-independence means that each statement may be compiled without regard to other statements. This, of course, is not possible within concurrent blocks, since we have to rule out the manipulation of the same data by two concurrent statements at the same time.

One may see that the ”common” parts of the compiler cause no problems, since they have already been implemented successfully many times. Once again the most difficult part is the code generator for software respectively net list generator for hardware targets.

2 Programming language

The semantics of the proposed language is based on the common hardware description languages whereas the syntax has been taken from C. At first glance the source code looks like "normal" C but the statements can have very different meanings (dependent on the context). As already mentioned, the language C has no construct to implement concurrency which is one of the most important characteristics of silicon devices. In C all statements are processed step by step. That means only one statement can be processed in a particular time slot. This is very typical for algorithms running on microprocessors (software implementations). The next problem we are faced with is the necessity to synchronize the concurrent statements. C does not provide such a mechanism. There is more than one possibility to implement this item (signals between concurrent statements; global clock etc.)

One can see that it is much easier to compile a "concurrent" program for the assembler code of a microprocessor. The concurrent items have to be serialized. The order of these items is arbitrary, since the data manipulated by them have no correlations; otherwise it is not possible to run them in a concurrent way.

2.1 Semantics

As already mentioned, it is much easier to serialize a parallel program than vice versa. Therefore it is a good choice to start with the semantics of a "parallel" description language; for example VHDL. Real VDHL is not suitable for our purpose, since it was originally developed for hardware documentation and therefore contains elements that are not synthesizable. In contrast, we want to propose a language that can be completely synthesized as well as compiled to microprocessor code.

2.1.1 Statements

The statements can be divided into two classes:

- *concurrent* statements
- *sequential* statements

Sequential statements cause no problems, neither in hardware nor in software. In software they correspond to the common sequence of machine commands that are processed step by step on a microprocessor, and in hardware they correspond to sequentially wired circuits. The concurrent statements do not cause any problems in hardware except for the synchronization of the different parallel data paths. They are mapped to parallel circuits. In software we have to distinguish two cases:

- The number of processors is greater than or equal to number of concurrent statements or
- the number is less

In most cases the number of processors is 1. Therefore the second case will occur more often. In the first case, each concurrent statement will run on one processor. This is faster than the second case, but it is necessary to synchronize the different processors. In the second case it is necessary to utilize a scheduler in order to run more than one statement on the same processor. This is the well-known method of running several tasks "quasi parallel" on one sequential processor. In the common case of having just one processor it is not necessary to synchronize the different concurrent statements, since the execution of one concurrent block is finished before the system goes on with the next block which uses the results of the current block as input. Therefore all the necessary data is ready at the same time (start of the execution of the succeeding block).

2.1.2 Data

This section looks into the question which data types/structures are supported. The types correspond to those known from the sequential programming language C. Even all the data structures are supported. They can be implemented in hardware and software. Unfortunately the language C does not support the data type "Boolean". This is always mapped to a byte or to another integer type. For microprocessor code this may be suitable, since the processor can process data types with bit length up to its register size in one cycle. When hardware circuits are used, this mapping is not suitable, since it consumes chip area without any sense (see Fig. 1). Therefore we add the type "bit" to the list of elementary data types. The declaration of variables with this data type is analogous to the declaration of any other variable of simple type. See the following code snippet.

Example 1:

```
{
   bit var0;
   //........
}
```

Without having this new type the compiler would produce the first circuit (Fig. 1). 7 of the eight wires (which correspond to a byte) are not necessary. In the second circuit the output of an if-statement was implemented with the new data type "bit". In this case only one wire is produced by the compiler (Fig. 2).
The language C provides four storage classes

- global variables (data segment)
- static local variables (data segment)
- automatic local variables (stack)
- dynamic variables (heap)

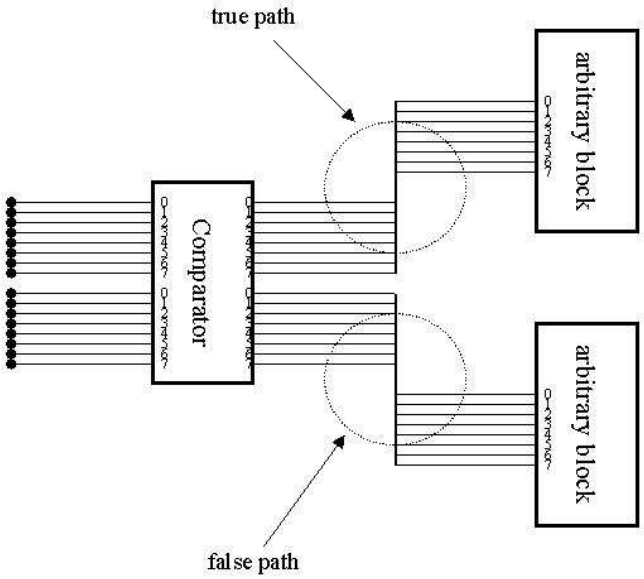


Figure 1: Comparator without data type 'bit'

The storage classes are mapped to the system in a totally different manner in hardware and software systems. Microprocessor-based systems store variables in 3 types of physical memory

- *registers*
- *RAM*
- *ROM*

The ROM contains the constants. They have fixed values during the whole life of the system. Variables are stored either in RAM or in one of the processor registers. The decision where they are stored is dependent on the duration between the production of a value and its usage as well as on the storage class. Global and static variables are always stored in RAM, whereas local variables and dynamic variables may be stored in registers. Anyway, the compiler tries to store as many variables as possible in the registers. This is much faster, since the manipulation of the values is always done in registers and it reduces the memory consumption.

If the source code is compiled to silicon the situation will be totally different. Nowadays most of the reconfigurable devices have an on-chip RAM. Therefore the variables which are stored in RAM in microprocessor systems can be stored similarly in silicon-based systems. But this

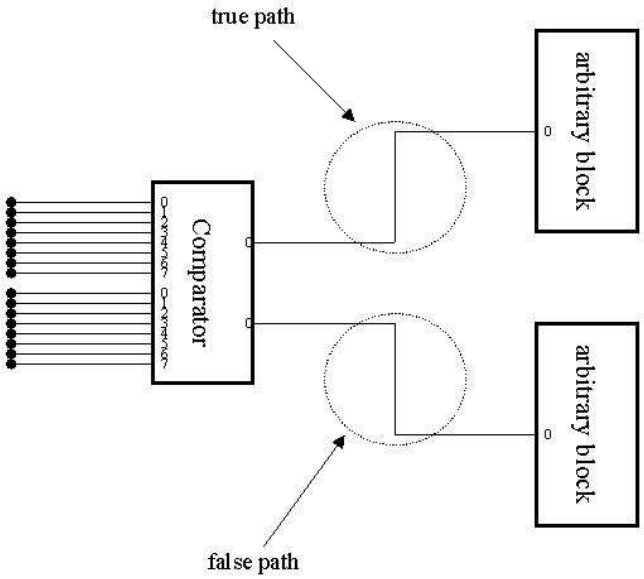


Figure 2: Comparator with data type 'bit'

makes no sense for local automatic variables. These variables will always be latched either at the outputs of the circuits which produce them or at the inputs of the "consuming" circuits. Which one is more suitable depends on the particular implementation.

2.1.3 Input/Output

In contrary to ANSI C we do not allow command line parameters, since they do not have meaningful equivalents in hardware. For data input and output we use external variable declarations with the prefix keyword "input" and "output" respectively.

Example 2:

```
input int x[100], y[100];
output int a, b;

void lin_regress(  int ndata )
{
   int i;
   int t, sxoss, sx=0, sy=0, st2=0, ss;
```

```
   b = 0;

   for (i=0; i<ndata; i++)
   {
      sx += x[i];
      sy += y[i];
   }
   ss = ndata;
   sxoss = sx/ss;

   for (i=0; i<ndata; i++)
   {
      t = x[i]-sxoss;
      st2 += t*t;
      b += t*y[i];
   }
   b /= st2;
   a = (sy-sx*b)/ss;
}
```

The example demonstrates the usage of input and output variables. This is indicated by the keywords "input" or "input" respectively. The program reads in a set of coordinates provided in the arrays x[100] and y[100]. It calculates the linear regression line and outputs the line parameters (a is the gradient and b is the the y-axis intercept).

The range of these variables is similar to all the externally defined variables. This is usually the whole program (except that they are declared with the keyword static; in this case they are limited to the file where they're declared). The user must declare an IO-channel for each of these variables, with the limitation that it is just allowed to read from the input channel or write to the output channel. Refer to the following examples.

Example 3:

```
input char a;
output char b;

void lin_regress( void )
{
   char chBuf;

   chBuf = a;
```

```
   if ( (chBuf >= 'a') && (chBuf <= 'z') )
      chBuf -= ('a' - 'A');

   b = chBuf;
}
```

Example 4:

```
input int a;
output int b;

void abs( void )
{
   char intBuf;

   intBuf = a;

   if (intBuf < 0)
      intBuf = -intBuf;

   b = intBuf;
}
```

The example above reads a character from an input stream and transforms it to a stream of capital letters if the input characters are small letters. The second part calculates the absolute value of an integer. It is absolutely necessary to buffer the values after the input, since one must not write to variables which are marked as input variables. This restriction is necessary, since the input variables are not real variables but they are input channels. Output variables have the same meaning. They are output channels. The programmer should write only final results to these variables, since every write request to these variables leads to an interaction with the peripherals which is suitable just for final values but not for intermediate values.

2.2 Syntax

The syntax is based on C. Unfortunately ANSI C is not suitable for hardware/software-codesign for the following reasons:

- C has no language elements to implement concurrency, which is one of the most import characteristics in hardware designs.
- C does not support time steering.
- C provides language constructs which have no immediate equivalent in hardware

- Some language features may not be compiled to hardware even if they are implementable (for example endless loops).

The language constructs can be classified into 3 groups :

- *assignments*
- *loops*
- *expressions*

2.2.1 Assignments

Assignments can occur in sequential as well as in parallel blocks. The usage of L-values in sequential blocks is not limited even if some constructs are not meaningful. See the following example (assume the following block contains only sequential statements).

Example 5:

```
{
   int x, y=1, z=2;

   x = y+z;
   x = 4;
}
```

The first assignment ($x = y + z$) is useless. The assignment ($x = y + z$) is done right before the second one ($x = 4$) and uses the same L-value (x). The value calculated in the first assignment can not be used at any time and is therefore senseless. Nevertheless, the block above is allowed. In contrary to sequential blocks it is not allowed to use the same L-value twice in one parallel block. See the following example.

Example 6:

```
{
   int x;

   par
   {
      x = 3;
      x = 4;
   }
}
```

The compiler is requested to assign the values 3 and 4 to the variable x at the same time. This, of course, is not possible.

There are no other restrictions on assignments compared to ANSI C.

2.2.2 Loops

Loops are the most difficult items in hardware/software co-design. Without considering the timing behaviour it is possible to implement all the types of loops known from ANSI C (for, do while, while). Even endless loops are possible in software implementations as well as in hardware implementations. In the assembler code of a microcontroller the loops are implemented using branch commands. The block of commands between the beginning and the end of a loop can be processed many times, with or without termination. Hardware circuits can be implemented similarly. The circuits which belong to the source code inside a loop may be activated many times, even endlessly. But for timing calculation purposes we restrict the implementation of loops if timing prediction is required. In this particular compilation mode, only a constant number of iterations are allowed. Timing prediction means that the compiler calculates the timing behaviour of the circuit or microcontroller system. See the following examples.

Example 7: (without timing prediction)

```
{
   int i, a;

   a = Get_a();

   for (i=0; i<a; i++)
   {
      //........
   }

   //........
}
```

Example 8: (with timing prediction)

```
{
   int i;

   for (i=0; i<10; i++)
   {
      //........
   }
```

```
   //........
}
```

The expression above has a constant number of loops. The compiler is able to calculate the loop processing time. The following examples are not allowed in this operation mode, since the processing time can not be calculated by the compiler.

Example 9:

```
{
   int i, a;

   a = Get_a();

   for (i=0; i<a; i++)
   {
      //........
   }

   for (i=a; i<10; i++)
   {
      //........
   }

   for (i=0; i<10; i+=a)
   {
      //........
   }

   //........
}
```

That means that all the R-values of the expression in the loop header must be constants. We allow just "for"-loops with a constant number of iterations.

In addition to "normal" for-loops we add a "parallel" for loop. This type may be used if the different loops are not correlated to each other. The syntax of this kind of loops is similar to the common "sequential" loop. The only difference is the keyword "par" instead of "for".

Example 10:

```
{
```

```
   int i, x[10];

   par (i=0; i<10; i++)
   {
      x[i] = i;
      //........
   }

   //........
}
```

The compiler can unroll the loop in order to make all the assignments at the same time in order to speed up the circuit. This, of course, is meaningful only if there is enough silicon space available.

2.2.3 Expressions

All the expressions known from normal ANSI C are supported. Refer to [11] for an entire description of ANSI C. Expressions are used to determine the value of a variable, inside of if statements etc. See a simple example below.

Example 11:

```
{
   int i, x[10];

   if (x[i]==i)
   {
      x[i] = 0;
      //........
   }
   else
   {
      x[i] -= 1;
      //........
   }

   //........
}
```

Notice that every expression always provides a result, even if it is not assigned to a variable and just used to determine the right path of a branch (like in the example above).

2.2.4 Additional timing construct

For timing behaviour calculations it is sometimes necessary to add some timing information to the IO-channels.

Example 12:

```
input int a, b;
output int c;
{
   int buf0, buf1;

   buf0 = a <0 milliseconds>;
   buf1 = b <0 milliseconds>;
   c = buf0 + buf1;

   //........
}
```

The example above reads in the values of the input variables a and b right after the block is activated. In this case no timing information is required.

Assume we want to calculate the sum of a and b 3 times with a latency time of 20 milliseconds in between. Then we write these times after the "read in" statement. The first pair of values of a and b is read right after the software or the silicon respectively is activated. But between the first and the second and the second and the third accesses to the IO-channel that belongs to these variables a delay of 20 seconds will be inserted. See the following example.

Example 13:

```
input int a, b;
output int c;
{
   int buf0, buf1;

   buf0 = a <0 milliseconds>;
   buf1 = b <20 milliseconds>;
   c = buf0 + buf1;

   //........
}
```

The values in brackets are absolute times. That means that the measurement starts when the silicon respectively the software is activated. It is also possible to measure the time relative

to the start time of the current block. Therefore the keyword *relative* has to be added to the timing statement. See the following example.

Example 14:

```
input int a;
{
   int cBuf[3];

   cBuf[0] = a; <0 milliseconds relative>
   cBuf[1] = a; <20 milliseconds relative>
   cBuf[2] = a; <40 milliseconds relative>

   //........
}
```

Of course, this algorithm may also be implemented by writing a loop with a delay after each calculation. But this is more difficult for the programmer, since he has to calculate the time of the addition and the loop administration and subtract this from the desired time (20 milliseconds in our example). See the following example.

Example 15:

```
input int a, b;
output int c;

{
   c = a+b;
   delay(20_MILISECONDS - calctime());

   //........
}
```

In some cases it is very difficult and maybe not important to calculate the absolute delay of the circuit. But, if a single bus is used to output the values, it is necessary to specify the output timing relatively to each other. For this purpose we introduce the timing keyword *relativeOut*. If the first value is ready for output, a special signal is activated to indicate this (refer to 3.10.3 for details of this signal). The timing value associated with the *relativeOut*-keyword is defined as the time passed after this signal has been activated. See the following example:

Example 16:

```
output int c;
```

```
{
   int cBuf[3];

   //........

   c = cBuf[0]; <0 clocks relativeOut>
   c = cBuf[1]; <2 clocks relativeOut>
   c = cBuf[2]; <4 clocks relativeOut>
}
```

The first value in the example above is output shortly after the output ready signal is activated. The second and the third one are output after two and three clock cycles respectively.
The unit (in our case "milliseconds") is not mandatory. The default value for all the timing expressions is "clocks". That means exactly one internal clock. For example.

Example 17:

```
c = a+b; <20>
```

is equivalent to

Example 18:

```
c = a+b; <20 clocks>
```

Allowed units are all the known time units

- *microseconds*
- *milliseconds*
- *seconds*
- *minutes*
- *hours*

and the additional unit

- *clocks*

2.3 Conclusion

An experienced C Programmer will recognize most of the elements in the proposed language. We have restricted the allowed elements even if it may be possible to build a hardware compiler which is able to translate much more constructs, for example

- dynamic memory management
- pointer arithmetic

But these elements rule out the possibility to predict the timing behaviour; or they can not be implemented efficiently in hardware. Even in software systems these constructs are a well-known source of errors.

A compiler for hardware/software co-design has been partially implemented; refer to section 3 for details. It produces code for ARM-processors as well as net lists. (represented by EDIF-files)

2.4 Program examples

The following examples implement a few simple mathematical algorithms. They are predestined for our purpose, since they are used in software as well as in hardware systems.

2.4.1 Linear transformation of an n-dimensional vector

A linear transformation in general may be represented by a matrix. Therefore we can write it as a multiplication of the corresponding matrix with the vector. Notice that there is exactly one matrix which represents a linear transformation. The following example performs a linear transformation in a 3-dimensional vector space.

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

A corresponding implementation in our C reads as follows.

Example 19:

```
#define DIMENSION 3

input int x;
output int y;

void linTrafo( void )
{
   int i;

   int xBuf[DIMENSION];
   int yBuf[DIMENSION];
```

```
   const int a[DIMENSION][DIMENSION] =
   {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
   }

   /* The input vector components are read
      sequentially. */
   xBuf[0] = x; <0 clocks>
   xBuf[1] = x; <1 clocks>
   xBuf[2] = x; <2 clocks>

   par (i=0; i<DIMENSION; i++)
   {
      yBuf[i] = a[i][0]*xBuf[0] + a[i][1]*xBuf[1] +
                a[i][2]*xBuf[2];
   }

   /* The output vector components are written
      sequentially. */
   y = xBuf[0]; <0 clocks relativeOut>
   y = xBuf[1]; <1 clocks relativeOut>
   y = xBuf[2]; <2 clocks relativeOut>
}
```

2.4.2 Matrix multiplication

Nest we show how the matrix multiplication

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

is implementated in our C.

Example 20:

```
#define DIMENSION 3

input int x;
output int y;
```

```
void matrMult( void )
{
   int i, j;

   int a[DIMENSION][DIMENSION];
   int b[DIMENSION][DIMENSION];
   int c[DIMENSION][DIMENSION];

   /* The components of input array a are read sequentially. */
   a[0][0] = x; <0 clocks>
   a[0][1] = x; <1 clocks>
   a[0][2] = x; <2 clocks>
   a[1][0] = x; <3 clocks>
   a[1][1] = x; <4 clocks>
   a[1][2] = x; <5 clocks>
   a[2][0] = x; <6 clocks>
   a[2][1] = x; <7 clocks>
   a[2][2] = x; <8 clocks>

   /* The components of input array b are read sequentially. */
   b[0][0] = x; <9  clocks
   b[0][1] = x; <10 clocks>
   b[0][2] = x; <11 clocks>
   b[1][0] = x; <12 clocks>
   b[1][1] = x; <13 clocks>
   b[1][2] = x; <14 clocks>
   b[2][0] = x; <15 clocks>
   b[2][1] = x; <16 clocks>
   b[2][2] = x; <17 clocks>

   par (i=0; i<DIMENSION; i++)
   {
      par (j=0; j<DIMENSION; j++)
      {
         c[i][j] = a[i][0]*b[0][j] + a[i][1]*b[1][j] +
                   a[i][2]*b[2][j];
      }
   }

   /* The output array components are written
      sequentially. */
```

```
   y = c[0][0]; <0 clocks relativeOut>
   y = c[0][1]; <1 clocks relativeOut>
   y = c[0][2]; <2 clocks relativeOut>
   y = c[1][0]; <3 clocks relativeOut>
   y = c[1][1]; <4 clocks relativeOut>
   y = c[1][2]; <5 clocks relativeOut>
   y = c[2][0]; <6 clocks relativeOut>
   y = c[2][1]; <7 clocks relativeOut>
   y = c[2][2]; <8 clocks relativeOut>
}
```

2.4.3 Linear regression

This method is used each time an experiment provides data samples with linear impact and the parameters of the corresponding line are required. In case of a straight line these are 2 constants: the intercept of the y axis and the gradient. The following formula minimizes the square distances of the samples to the regression line; refer to [14] for details.

$$\sum_{i=1}^{n}(y_i - (a + bx_i))^2 = Min \tag{1}$$

A corresponding implementation in our C reads as follows.

Example 21:

```
input int x_in, y_in;
output int a_out, b_out;

void lin_regress( void )
{
   int x[10], y[10];
   int i;
   int t, sxoss, sx=0, sy=0, st2=0, ss;
   int a, b;

   b = 0;

   x[0] = x_in <0 clocks>;
   x[1] = x_in <1 clocks>;
   x[2] = x_in <2 clocks>;
   x[3] = x_in <3 clocks>;
   x[4] = x_in <4 clocks>;
   x[5] = x_in <5 clocks>;
```

```
   x[6] = x_in <6 clocks>;
   x[7] = x_in <7 clocks>;
   x[8] = x_in <8 clocks>;
   x[9] = x_in <9 clocks>;

   y[0] = x_in <10 clocks>;
   y[1] = x_in <11 clocks>;
   y[2] = x_in <12 clocks>;
   y[3] = x_in <13 clocks>;
   y[4] = x_in <14 clocks>;
   y[5] = x_in <15 clocks>;
   y[6] = x_in <16 clocks>;
   y[7] = x_in <17 clocks>;
   y[8] = x_in <18 clocks>;
   y[9] = x_in <19 clocks>;

   for (i=0; i<10; i++)
   {
      sx += x[i];
      sy += y[i];
   }
   ss = ndata;
   sxoss = sx/ss;

   for (i=0; i<ndata; i++)
   {
      t = x[i]-sxoss;
      st2 += t*t;
      b += t*y[i];
   }
   b /= st2;
   a = (sy-sx*b)/ss;

   a_out = a;
   b_out = b;
}
```

The implementations above are not optimal but they demonstrate efficiently the usage of our programming language. Especially the first 2 examples may be implemented differently in real applications.

Refer to section 5 for some experiments with the proposed programming language.

3 Compiler

One may see that the compiler has to branch between hardware and software compilation at the latest when the grammar is analyzed. In the case of a hardware-target the language is context-dependent, as already mentioned, whereas it may be seen as almost context-independent if a microcontroller target is desired. Please notice, that even ANSI C is not fully context-independent; i.e. a variable has to be defined earlier in the text before usage (refer to [12] for details).

One may see that the standard parts of the compiler cause no problems, since they have already been implemented successfully many times. Once again the most difficult part is the code generator for software and the net list generator for hardware targets.

3.1 Preprocessor

Common C compilers use a preprocessor for the program script. The preprocessor is very helpful to manage fragmented source codes and steer the compilation process. Anyway the preprocessor may differ in details from one compiler to another. We have decided to implement the preprocessor as described in the C primer by Kernighan and Ritchie (refer to [11]).

3.2 Compilation

As mentioned in the introduction, the compilation process is done in several steps (in this context, the preprocessing is not viewed as part of the compilation, since it is just a kind of text processing of the source code).

We've split our compiler into 4 main blocks. This partition is similar to the majority of the publications.

- source code analysis
- intermediate code generator
- optimization and timing analysis
- code or net list generator

Since our compiler is able to build microprocessor code and net lists from the same source code, it is necessary to branch for one of the targets at a particular point. We have decided to make this branch within the block called 'optimization and timing analysis'. Some optimization techniques may be used for both targets. But there are also highly target-dependent techniques. We decided to put both parts to one block to have a better overview.

Source code analysis is probably the easiest part of the software, since it has been implemented several times for common sequential compilers. It may be developed from scratch as a 'normal' program, but the more efficient way is to use the tools called 'lex' and 'yacc' which are available for free on almost all UNIX systems. We have decided to use lex and yacc to implement the source code analysis. The analysis is done in two steps. The first one splits the input character stream into a list of strings (called lexemes). This is done by the tool called lex. Then yacc reads these lexemes and sorts them in a syntax tree according to predefined rules. Refer to [12] for further explanations of syntax trees.

3.2.1 Intermediate code generator

The intermediate code is a common low-level base for both targets. The intermediate code generator reads the syntax tree (produced by yacc) in reverse direction and generates this code. As mentioned it is a low-level code for a sequential machine. The syntax of the proposed intermediate code is proprietary and will not be found in the literature. See the following example.

Example 22:

```
int a, b;
int x[100], y[100];

void lin_regress(  int ndata )
{
   int i;
   int t, sxoss, sx=0, sy=0, st2=0, ss;

   b = 0;

   for (i=0; i<ndata; i++)
   {
      sx += x[i];
      sy += y[i];
   }
   ss = ndata;
   sxoss = sx/ss;

   for (i=0; i<ndata; i++)
   {
      t = x[i]-sxoss;
      st2 += t*t;
      b += t*y[i];
   }
```

```
   b /= st2;
   a = (sy-sx*b)/ss;
}
```

The compiler generates the following intermediate code.

Example 23:

```
lin_regress
===========
(0)     (=) st2 <-- [0]
(1)     (=) sy <-- [0]
(2)     (=) sx <-- [0]
(3)     (=) b <-- [0]
(4)     (=) i <-- [0]
(5)     (<) CCHW_TMPVAR_16 <-- [i ndata]
(6)     (if) [CCHW_TMPVAR_16 11]
(7)     (+) sx <-- [sx x,i]
(8)     (+) sy <-- [sy y,i]
(9)     (++) i <-- [i]
(10)    (B) [5]
(11)    (=) ss <-- [ndata]
(12)    (/) sxoss <-- [sx ss]
(13)    (=) i <-- [0]
(14)    (<) CCHW_TMPVAR_25 <-- [i ndata]
(15)    (if) [CCHW_TMPVAR_25 23]
(16)    (-) t <-- [x,i sxoss]
(17)    (*) CCHW_TMPVAR_30 <-- [t t]
(18)    (+) st2 <-- [st2 CCHW_TMPVAR_30]
(19)    (*) CCHW_TMPVAR_32 <-- [t y,i]
(20)    (+) b <-- [b CCHW_TMPVAR_32]
(21)    (++) i <-- [i]
(22)    (B) [14]
(23)    (/) b <-- [b st2]
(24)    (*) CCHW_TMPVAR_37 <-- [sx b]
(25)    (-) CCHW_TMPVAR_36 <-- [sy CCHW_TMPVAR_37]
(26)    (/) a <-- [CCHW_TMPVAR_36 ss]
(27)    (RFS)
```

One can see that the semantics is already similar to the assembler of a common RISC processor. The operators in brackets have to be replaced by the assembler mnemonics and the symbols have to be mapped to registers. The last step of course, is not trivial, since the number of

available registers is limited. Consider as example the intermediate code line:

Example 24:

```
(7)     (+) sx <-- [sx x,i]
```

This statement adds the value of the variable x to the offset determined by i of the variable x. One can see that x may not be a simple variable but a data structure. Otherwise the base/offset addressing mode does not make sense. The result of the operation is stored in the variable sx. This intermediate statement may be translated easily to the assembler code of a RISC processor. If we assumed to utilize an ARM compatible processor the assembler code will look as follows:

Example 25:

```
LDR R7,[R8,R9]   ;load the value of x, offset i to register R7
ADD R5,R6,R7     ;add the value of x, offset i to sx and store it in R5
```

The short example above shows that the structure of the produced intermediate code is very similar to the assembler mnemonics of a current Risc-machine, but of course, much more abstract. The syntax of the intermediate code is quite simple:

```
<operator> [<L-value> <operand1> <operand2>]
```

The objects in *[]*-brackets depend on the operator. At least an operator for each statement is required. Therefore the simplest possible intermediate code statement is the following:

```
<operator>
```

The only currently implemented statement of this simple type (without any additional operand) is the return from subroutine. Refer to statement (27) of the code snippet in example 23.

The operator is one of the following options: ("+" "-", "*", B(Branch), RFS (Return from subroutine...). The operators/statements are divided into two classes. The first class contains control statements whereas the second class contains operators which belong to the data path. The control statements are listed below:

- *B* branch
  The branch operator has one parameter that indicates the statement index to branch to. See the following example (extracted from example 23):

  Example 26:

  ```
  (10)   B   [5]
  ```

The execution branches immediately to the statement with index 5.

Example 27:

```
                ......................
(5)             (<) CCHW_TMPVAR_16 <-- [i ndata]
(6) - (9)       ......................
(10)            (B) [5]
                ......................
```

In this particular example the branch statement belongs to a loop. The loop is repeated until the *CCHW_TMPVAR_16* is *0*.

- *if* conditional branch The *if*-statement has two operands. The first one is the variable that contains the truth value and the second one is the start index of the false path. Refer to the following example:

  Example 28:

  ```
  (6)     (if) [CCHW_TMPVAR_16 11]
  ```

  If the value of *CCHW_TMPVAR_16* is *0* then the engine resumes operation at index 11. If the value of *CCHW_TMPVAR_16* is unequal *0* then the execution goes on at index 7 directly after the *if*-statement.

- *RFS* return from subroutine This operator returns to the caller's module.

- *BS* branch subroutine The *BS* operator invokes a subroutine. There are two types of *BS* statements; with and without a return value (corresponds to a void function in C). The first one has an L-value at the left side of the arrow (Example 29) whereas the second one has no L-value, even no arrow sign (Example 30).

  Example 29:

  ```
  (2)   (TS) 3 <-- [2]   (BS)   det1 <-- [(buf_00,buf_01,buf_10,buf_11)]
  ```

  Example 30:

  ```
  (2)   (TS) 3 <-- [2]   (BS)   [(buf_00,buf_01,buf_10,buf_11)]
  ```

  The objects in the brackets *[(...)]* are passed to the subroutine. Their number is arbitrary (even 0 objects are allowed). A subroutine with 0 input parameters or no return value seems to be senseless at first glance. But of course, it can communicate directly with the external world using *input/output* channels. Refer to Fig. 3.

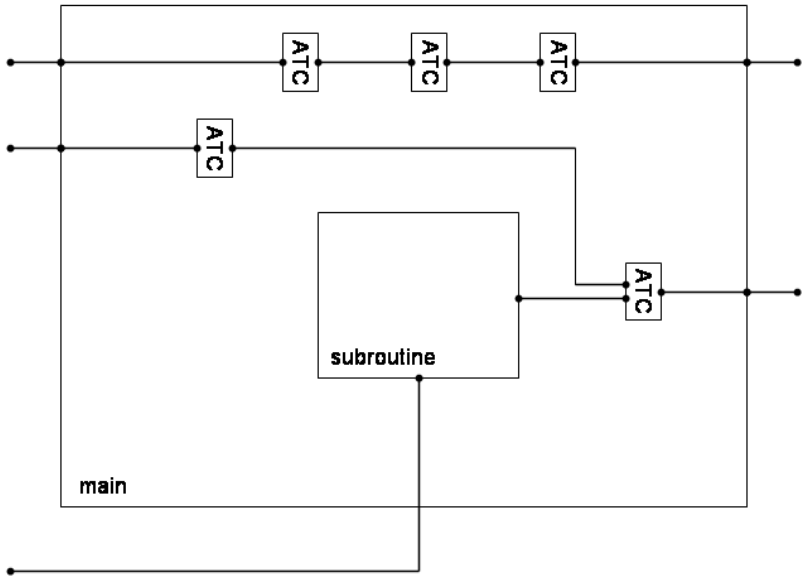


Figure 3: Function call without parameters

The intermediate code supports concurrent execution. Each line consumes exactly one clock cycle (top to bottom). The cycle index is on the left side of each line. Refer to the following source code example. The source code block of concurrent commands is prefixed by the keyword *par*.

Example 31:

```
void lin_regress(  int ndata )
{
   int i;
   int t, sxoss, sx, sy, st2, ss;

   //........

   par
   {
      sx = 0;
      sy = 0;
      st2= 0;
      ss = 0;
   }

   //........
}
```

The intermediate code generator produces the following code:

Example 32:

```
lin_regress
===========
(0)   (=) sx <-- [0]   (=) sy <-- [0]   (=) st2 <-- [0]   (=) ss <-- [0]
(1)   (RFS)
```

One can see that the assignments within the *par* block is executed concurrently. These assignments of course, are serialized if the final target is not able to execute commands concurrently (i.e. a single processor engine). Please refer to appendix 3.9 for a full description of the intermediate code commands.

3.3 Optimization

We have to distinguish between two goals of optimization:

- for speed
- for memory or for silicon consumption respectively.

In general the two goals can not be achieved at the same time, that means, if we want to increase the speed, we have to take into account that the memory/silicon requirement is higher and vice versa.

Many approaches for optimization at different levels of abstraction have been published in the past years. Some of them are utilized in our compiler. The optimizer described in this paragraph is focused on the intermediate code even if some minor optimizations are done at the source code level and at the assembler code or net list level.

The optimization of the intermediate code may roughly be divided into two subsections. The first one is common for both target types whereas the second one is highly dependent on the desired target type (microprocessor code or hardware net lists).

3.3.1 Optimization (first step)

In C as well as in all other context-independent grammar languages the programmer is allowed to write any kind of statement even if they are not useful or at least their order in the program is not meaningful.

Example 33:

```
{
   int a, b, c;
```

```
   a = b;
   c = a;
}
```

If the variable a is not used anymore in subsequent statements it can be removed and the two statements are replaced by just one. Normally the statements are not as simple as in the example above. Therefore it is more efficient to do these optimizations with the intermediate code. See example below.

Example 34:

```
UINT8 main( UINT8 inp0, UINT8 inp1, UINT8 inp2, UINT8 inp3 )
{
   UINT8 buf;

   buf = inp0 + inp1 + inp2 + inp3 + inp4 + inp5;

   return buf;
}
```

A straightforward compilation (without compilation) generates the following intermediate code:

Example 35:

```
main
====
(0)   (TS) 1 <--|      inp0 <--|   inp1 <--|   inp2 <--|   inp3 <--|
                       inp4 <--|   inp5 <--|
(1)   (TS) 2 <-- [1]   (+) CCHW_TMPVAR_17 <-- [inp0 inp1]
(2)   (TS) 3 <-- [2]   (+) CCHW_TMPVAR_16 <-- [CCHW_TMPVAR_17 inp2]
(3)   (TS) 4 <-- [3]   (+) CCHW_TMPVAR_15 <-- [CCHW_TMPVAR_16 inp3]
(4)   (TS) 5 <-- [4]   (+) CCHW_TMPVAR_14 <-- [CCHW_TMPVAR_15 inp4]
(5)   (TS) 6 <-- [5]   (+) buf <-- [CCHW_TMPVAR_14 inp5]
(6)   (TS) 7 <-- [6]   (=) RFS_VAR_NAME <-- [buf]
(7)   (TS) |<-- [7]    (RFS) [RFS_VAR_NAME]
```

It is evident that this intermediate code is not optimal. An additional value is added to the temporary variable each clock. The parallelization of the statements is not possible in this case. The best way to optimize this code is to cluster the additions as done in the next example.

Example 36:

```
main
====
(0)   (TS) 1 <--|      inp0 <--|   inp1 <--|   inp2 <--|   inp3 <--|
                       inp4 <--|   inp5 <--|
(1)   (TS) 2 <-- [1]   (+) CCHW_TMPVAR_15 <-- [inp4 inp5]
                       (+) CCHW_TMPVAR_17 <-- [inp2 inp3]
                       (+) CCHW_TMPVAR_16 <-- [inp0 inp1]
(2)   (TS) 3 <-- [2]   (+) CCHW_TMPVAR_14 <-- [CCHW_TMPVAR_16 CCHW_TMPVAR_17]
(3)   (TS) 4 <-- [3]   (+) buf <-- [CCHW_TMPVAR_14 CCHW_TMPVAR_15]
(4)   (TS) 5 <-- [4]   (=) RFS_VAR_NAME <-- [buf]
(5)   (TS) |<-- [5]    (RFS) [RFS_VAR_NAME]
```

One can see that this intermediate code runs faster than the non-optimized version. Refer to [12] for details on this kind of optimizations.

3.3.2 Optimization (second step)

The second step is highly dependent on the target class. It takes care of the particular features of software systems or hardware devices respectively. State-of-the-art microprocessors provide a lot of registers. The majority of them are for general purpose and may be used to store and manipulate the values of variables. Statements may be reordered in terms of their execution schedule in order to avoid storing and reloading the values to/from the RAM. The following example shows how statement reordering reduces the RAM access:

Example 37:

```
(1)     (TS) 2 <-- [1]          (+) buf0 <-- [inp0 inp1]
(2)     (TS) 3 <-- [2]          (+) buf1 <-- [inp2 inp3]
(3)     (TS) 4 <-- [3]          (+) buf2 <-- [buf0 inp4]
```

Assume that there are no more registers available when the statement (2) is to be executed. In this case the value of *buf0* must be copied to RAM. Switching the first two statements avoids this time-consuming action.

Another option to increase the speed of a microprocessor is the opcode pipelining. That means that an opcode is fetched, decoded etc. on a pipeline within the processor. But if the processor reaches a branch it has to reload the pipeline, since it knows the operation type just after decoding it. The optimizer has the chance to optimize the code in order to use as few branches as possible. On the other side, hardware devices are able to execute statements concurrently. The more statements are executed concurrently the faster is the overall execution time.

Another aspect is the input latency. That means the time after reading input data when the circuit is blocked for other inputs.

3.4 Timing analysis

One of the most interesting aspects of our work is the prediction of the timing behaviour. Conventionally the timing analysis is done experimentally with the finished design. If the result does not meet the requirements the developer will go back into the design chain and refine the design. Our approach is totally different. We try to use the intermediate code for timing prediction. Please refer to section 4 for details on this item. Please notice that a clock-accurate timing is possible only in hardware implementations.

3.5 Code/circuit generation

The generation of microprocessor code and hardware circuits are totally different tasks. The generation of microprocessor code from a sequential programming language like C is a well-known task. Compilers have been implemented for lots of processor architectures (Cisc, Risc, DSP etc.). For that reason we focus our interests on the generation of net lists of logic gates. The routing and mapping of the logic gates to particular devices cause no problems. Tools for routing and mapping are provided by the device vendors for free (for example "Quartus" by Altera Inc. or "ICE" by Xilinx). Therefore we stop our discussion at the gate level.

3.5.1 Microprocessor code

Our compiler generates code for the ARM7 core. This core is incorporated by a majority of microcontrollers for embedded devices. Also a core simulator (ARMULATOR) is available for debugging purposes on Windows platforms.

The ARM7 incorporates 13 general purpose registers and a lot of steering and control registers (stack pointer, link register etc.). The steering and control registers have their particular purpose. Their usage follows directly from the intermediate code. For example the link register is used to store the return address before a function is invoked. The stack pointer register points to the top of the stack. It is incremented or decremented each time the stack is pushed respectively popped. More interesting is the usage of the general purpose registers. These registers are used to store and manipulate the variables. The problem is to map the variables to registers efficiently; that means, to exchange the variables values between the memory and the register as rarely as possible. We have implemented a statistic method for this task. Therefore we introduce a function $f^k(t)$ which defines the priority of a variable within a functional block:

$$f^k(t) = \begin{cases} \infty : if\ k\ is\ not\ used\ \forall t' > t \\ A * (t - t_E) + B * (t_N^k - t) + C_L(k,t) + C_R : else \end{cases}$$

t measures the time (number of elapsed clock cycles) from the start of a particular function.

A, B and C_L are proportional constants. t_E is the end time of the examined function and t_N the time of the next appearance of the variable k. The constant $C_L(k,t)$ is added if the variable k

is used within a loop at the time t. If at a particular time t^* no registers are available, a register is captured from the variable with the lowest priority ($\min_k f^k(t^*)$), and its value is transferred to the RAM. C_R is a value which is added if the variable is declared in the source code with the keyword "register". Most intermediate code objects can be translated straightforwardly to the ARM7 commands (refer to [8] for the complete instruction set of the ARM7). Some command cause a little bit more work, since the ARM7 does not provide a useful instruction (for example the arithmetic division).

3.5.2 Digital circuits

We want to focus our interests on this item, since the compilation of C to assembler code for a sequential microprocessor is a well-known task.

We assume that the optimization of the intermediate code is finished at this point. The task is to translate the commands of the intermediate code to a lower level of representation. A suitable level of abstraction are net lists of logic gates. The last step after this process is the routing and placing, but we do not need to take care of this final process for two reasons: The final step is identical to classic ways of hardware development (hardware definition language of sketch editors). This part is highly device-dependent and the number of vendors for these devices is high.

Complex algorithms can be translated efficiently only into clocked circuits. The data is processed within the circuits step by step. Therefore the entire circuit is split into pieces of logic gates. Each of these blocks is processed at a particular time and activated by an external clock (see the sketch below; Fig. 4).

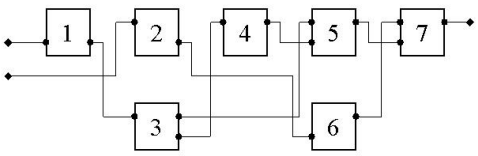


Figure 4: Clocked circuit

The example in Fig. 4 contains 7 blocks. It takes 5 clock cycles for execution. Blocks 2,3 and 5,6 are executed within the same clock cycle. This, of course, is possible only if different data are manipulated by these blocks. It is evident that the topology of these blocks has a strong impact on the quality of the circuit. Refer to Chapter 4 for details on this item.

It is possible to map each of the commands in the intermediate code to one block. For efficient circuits this is not suitable, since the number of subsequent logic gates is different for different types of commands. For example a multiplication may be built of a sequence of additions (a 8-Bit multiplication is built of 7 subsequent additions; see Fig. 5).

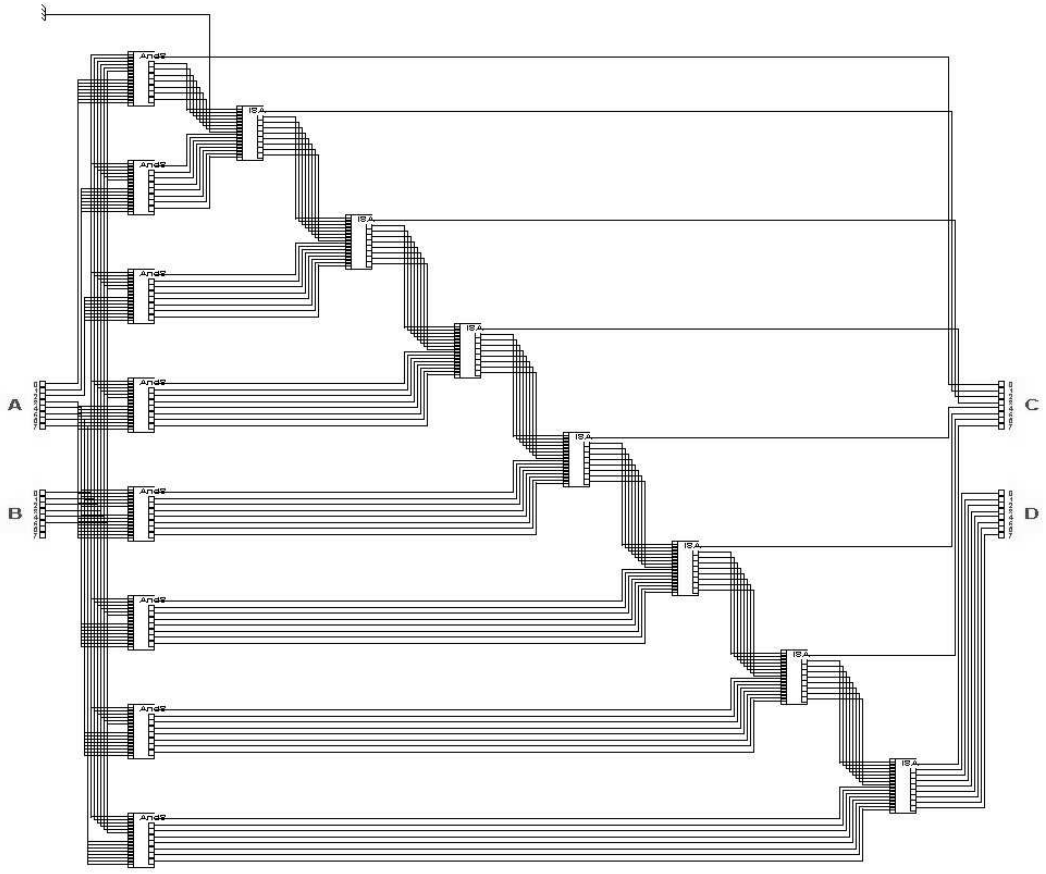


Figure 5: 8-Bit multiplier

If we implement our system such that all the commands take the same time, we will need to latch the result of a fast command for a long time. For example, the circuit delay of a multiplication is seven times the delay of an addition (since the circuit is built of seven sequential additions). For this purpose we model formally the multiplication by 7 linked blocks, each of them taking one cycle (Fig. 6).

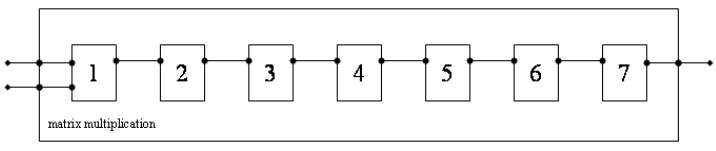


Figure 6: Model of a multiplier by 7 sequentially linked blocks

In the model above the matrix multiplication ATC is split into seven blocks. From a timing point of view the seven ATCs are independent. But of course, the chronology of the new blocks is fixed, since the different ATC have fixed data links. If we use this model it will be possible to increase the execution clock, since the maximum delay time of each ATC is the same as the delay time of an addition.

Systems currently available on the market (for example the DK1 suite from Celoxica Inc) assume that all the statements at the source code level take the same execution time. This assumption enables the user to calculate the execution time easily. But it forces the programmer to take care of the execution time when writing each statement in the source code. In this case it may be a bad habit to mix short and long statements. In our system the way of writing source code statements has no impact on the execution time. Refer to section 6.2.2 for a detailed discussion of this item.

3.6 Atomic commands

Our compiler provides a default circuit implementation for each of the intermediate code commands. Hence we call them atomic commands. These implementations are generic and not optimized for a particular target. The level of abstraction for this mode is structural at the gate level. See Fig. 7 (it is our default implementation for the addition). The source code for this block may be found in the appendix 3.8 together with some other examples.

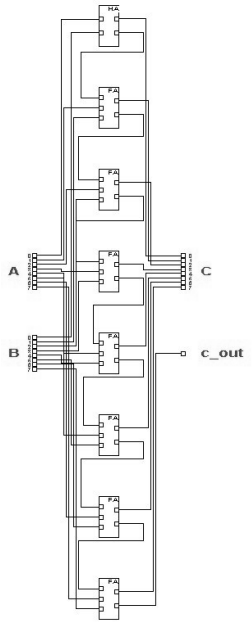


Figure 7: 8-Bit adder

Different implementations for all the used commands in the intermediate code may be found in the literature; refer to [15].

3.7 Some compiler details

The compiler has not been finished yet. The current status demonstrates that our approach is realizable and the results are sensible.

3.7.1 Calling conventions

The compiler is implemented as a console application for WIN32 platforms. There are default values for all the command line options. Therefore the simplest way to run the program is calling the executable together with a source code file.

Example 38:

```
{
   c:>...\bin\cchw source.hc
}
```

The options (if required) are written between the program name and the source-code file name.

Example 39:

```
{
   c:>...\bin\cchw [options] source.hc
}
```

The following options are supported:

- -t directory name *directory where the intermediate files are stored*
- -v *verbose mode*
- -I directory name *directory where the include files are searched*
- -p *generate an output after running the preprocessor*
- -a ATC file *add the ATC file to the list of file names*
- -i *interactive mode*
- -c *don't link the compiled objects (stop after compilation)*
- -S *don't link the assembled objects (stop after assembly)*
- -trg [target] *desired target:*
  - NL *net list*
  - ARM7 *ARM7 microcontroller code*
  - ATC *net list for atomic commands*

- -o[digit] *level of optimization (0-9)*

Below there are the default values for the command line options:

- -t ./ *local directory*
- -v *none*
- -I *none*
- -p *none*
- -a *empty list*
- -i *none*
- -c *none*
- -S *none*
- -trg ARM7
- -o0 *no optimization*

The name of the output file is generated from the name of the input files. It's extension is dependent on the options. See below the list of output file extensions.

- .s *ARM assembler file*
- .edf *EDIF file for net lists*
- .hcp *preprocessed source code*
- .enl *internal net list format; used also for graphical output*
- .egl *internal ATC net list format; used also for graphical output*
- .o *ARM object file*
- .m32 *Motorola 32 Bit S-records*

Notice that not all the file types are supported in the current version.

3.8 Implementation of atomic commands

In this section we explain the usage of the structural description[10] which is used to implement atomic commands. The syntax is taken from C but the semantics is totally different. The implementation of algorithm in ANSI C is always behavioural. As already mentioned the atomic commands are implemented using a structural description. Therefore the main operator is the right arrow (→). Hence we call it the link operator. It links two objects with a wire. The only predefined objects are:

- *NOT*; one input; invert the logic input state
- *AND*; two inputs; output is logic high if both inputs are high, else low
- *OR*; two inputs; output is logic low if both inputs are low, else high

The three objects above are the basic bool functions *not*, *and* and *or*. They are used to declare digital gates (the syntax is similar to the declaration of variables in C) which are linked together with the → operator.

3.8.1 Semantics of the ATC implementation

The semantics is quite simple. Each object is defined similar to functions in C.

```
<object name> <output list> ( <input list> )
```

The object name is arbitrary. The output list declares the output wires of the object whereas the input list (in *()*-brackets) declares the input wires.

output list:

```
output wire0, output wire1, ..., output wire n
```

input list:

```
input wire0, input wire1, ..., input wire n
```

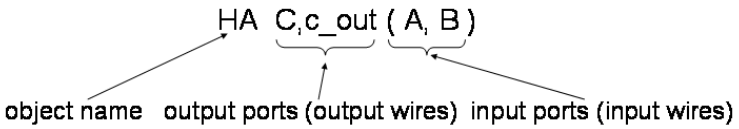


Figure 8: user defined ATC declaration

The input and output wires may also be declare as port bundles. This is done similar to the array declarations in C. The number in *[]*-brackets is the number of wires in the bundle. The

[10] See [16] for details on the structural and behavioural descriptions

access to a particular wire within the bundle is also done similar to the access to the component of an array. See the following example:

Example 40:

```
gateX C[8],c_out ( A[8], B[8] )
{
..............
   A[0] --> C[1];
}
```

The example above defines two input port bundles named A, B, one output port bundle named C that contain eight wires each and one single wire output port named *c_out*. The first wire in the input bundle A is connected to the second wire in the output bundle C.

3.8.2 Adder for 8bit integers

The following two examples implement a half-adder (HA) and a full-adder FA. In addition to the three basic gates all the user defined objects may also be used to declare gates. For example the half-adder implemented in the first example is used to implement the full-adder.

Example 41:

```
// The following
HA C,c_out ( A, B )
{
   { // declare two inverters
      NOT inp_Invert_A, inp_Invert_B;
   }
   { // declare three AND gates
      AND And0, And1, c_And;
   }
   { // declare an OR gate
      OR out_Collect;
   }

   // connect the inputs to the invertors
   A --> inp_Invert_A;
   B --> inp_Invert_B;

   // connect the inverted input A to the first port and
   // the input B to the second port of an AND gate
   inp_Invert_A --> And0[0];
```

```
   B --> And0[1];

   // connect the input A to the first port and
   // the  inverted input B to the second port of another AND gate
   A --> And1[0];
   inp_Invert_B --> And1[1];

   // connect the input A to the first port and
   // the input B to the second port of another AND gate
   A --> c_And[0];
   B --> c_And[1];

   // collect the outputs (connect to an OR gate)
   And0 --> out_Collect[0];
   And1 --> out_Collect[1];

   // connect the collected outputs to the output of the half-adder
   out_Collect --> C;

   // connect the carry to the carry output of the half-adder
   c_And --> c_out;
}

FA C,c_out ( c_in, A, B )
{
   { // declare a half-adder
      HA ha0, ha1;
   }
   { // declare an OR gate
      OR out_Collect;
   }

   // The following statements are the typical wiring of a full-adder
   c_in --> ha0.A;
   A --> ha1.A;
   B --> ha1.B;

   ha1[0] --> ha0.B;
   ha0[1] --> out_Collect[0];
   ha1[1] --> out_Collect[1];
```

```
   // connect the collected outputs to the output of the full-adder
   ha0[0] --> C;

   // connect the carry to the carry output of the full-adder
   out_Collect --> c_out;
}
```

A 8-Bit adder may be implemented similar using eight full adders. A graphical output of the 8-Bit adder may be found in Fig. 7.

3.9 The complete list of intermediate commands

The following intermediate commands are internal. They are optimized with the goal of implementing an optimizer and a generator for microprocessor code as well as generator of silicon gates.

- *if* followed by 3 operands. The result is 1 operand.
- = followed by 2 operands. Assign value of operand 2 to operand 1.
- + followed by 3 operands. The result is the sum of operand 2 and 3. The result is stored in operand 1.
- − followed by 3 operands. The result is the difference of operand 2 and 3. The result is stored in operand 1.
- ∗ followed by 3 operands. The result is the product of operand 2 and 3. The result is stored in operand 1.
- / followed by 3 operands. The result is the quotient of operand 2 and 3. The result is stored in operand 1.
- & followed by 3 operands. The result is the logical bitwise AND of operand 2 and 3. The result is stored in operand 1.
- | followed by 3 operands. The result is the logical bitwise OR of operand 2 and 3. The result is stored in operand 1.
- < if the value of operand 2 is lower than operand 3 the result is 0, otherwise it is 0. The result is stored in operand 1.
- > followed by 3 operands. If the value of operand 2 is greater than operand 3 the result is 0, otherwise it is 0. The result is stored in operand 1.
- <= followed by 3 operands. If the value of operand 2 is lower or equal than operand 3 the result is 0, otherwise it is 0. The result is stored in operand 1.

- >= followed by 3 operands. If the value of operand 2 is greater or equal than operand 3 the result is 0, otherwise it is 0. The result is stored in operand 1.
- ! = followed by 3 operands. If the value of operand 2 is not equal than operand 3 the result is 0, otherwise it is 0. The result is stored in operand 1.
- == followed by 3 operands. If the value of operand 2 is equal than operand 3 the result is 0, otherwise it is 0. The result is stored in operand 1.
- ++ followed by 2 operands. The operands are the same symbols. The value is incremented by one.
- −−followed by 2 operands. The operands are the same symbols. The value is decremented by one.
- *B* followed by 1 operand. Branch to the command with time stamp determined by the operand.
- *RFS* Return from subroutine.
- *BS* Branch to subroutine.

3.10 Linker

The linking process is the very last task of the compilation process. It compiles all the different modules and generates an executable output file. This is at least the imagination of software developers. The executable is a binary file that may be downloaded to a processors work space and started either by power up or by a command in an OS shell. The situation is totally different if a hardware target is selected. Even in this case the linker may be the last stage (i.e. the design will run on a logic simulator) but the result may also be post-processed if a real hardware device is generated. In this case the formal description of the circuit (net list file) has to be mapped to the physical structure of the particular target. Refer to Fig. 9.
Anyway our compilation process stops after the linking process. The example shown in Fig. 9 selects a Xilinx[11] device as hardware target. The post-processing (placing[12] and routing[13]) is done by the ISE refer to [18] for details) tool provided for free by Xilinx.

3.10.1 Software target

The compiler output is assembler code if the software target is selected. The assembler code may be processed by the assembler tool taken from the ARM developer suite [9] or the SDT [10]. Since the compilation process for software targets stops before assembly it is evident that a linker has not been implemented. Assembly and linking are quite simple tasks and not within the scope of this thesis.

[11] one of the two market leading FPGA vendors
[12] map the logic cells to the lookup tables of the FPGA
[13] connection between the logic cells

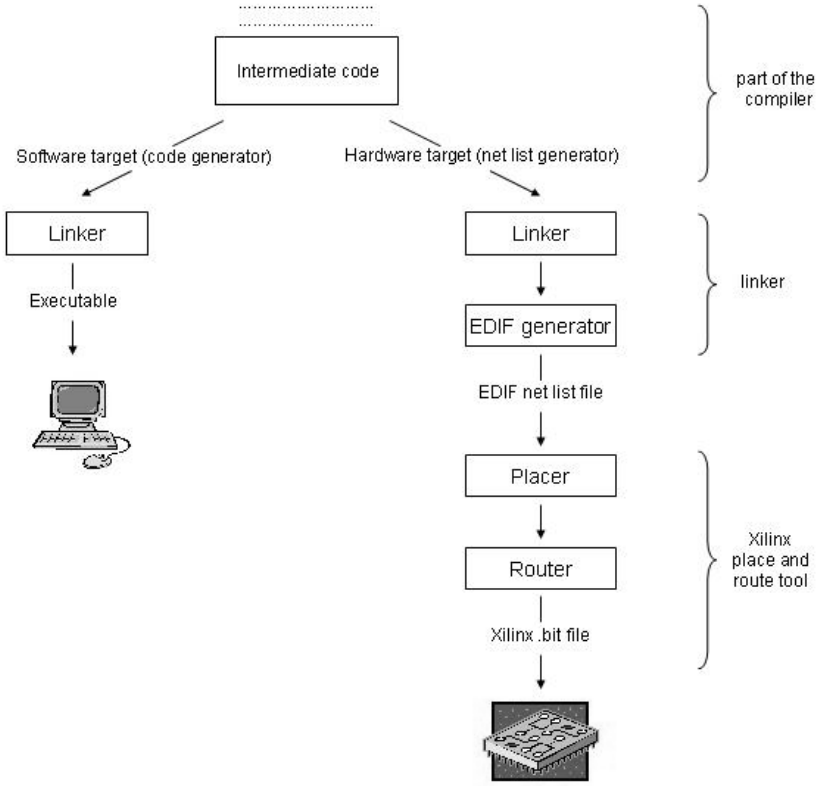


Figure 9: Linking process

3.10.2 Hardware target

The compiler output for hardware targets is a internal representation of the net lists stored in files with the extension *.enl* followed by an ordinary number that represents the appearance of the function within the intermediate code. The compiler generates one output file for each function. Refer to the intermediate code in example 50. This code contains four functions. For example the net list of function 3 will be stored in a file named *matr2Mult.enl3*. The structure of the net list file is quite simple. See the following C structures for details.

```
{
   typedef struct
   {
      char KeyWord[MAX_NAME_LENGTH];  // file Keyword
      T_ClockInfo globalClock;  // clocking information
      int fct_symIdx;  // index of function within the symbol table
      int fct_entries_Nr;  // number of entries
      T_atm_cmd ATCs[ENTRIES_NR][ATCs_Nr];  // all the ATCs
   }
   T_propNetList;
}
```

The magic word is always *Schem*. Refer to appendix 9.5 for the meaning of the other components.

The linker output is an EDIF[14]-file. Our EDIF file is a low-level representation of the entire circuit. The external variables (input/output variables and function parameters) and function names within the source code may be recognized in the EDIF files, since these arbitrary names are generated from the source code names. For example the function *matr2_Det* may be found in the EDIF file under the same name within the library *libmain*. Each function generates a cell in the EDIF file and each parameter generates a port or a port bundle in the cells header. Refer to the following example:

Example 42:

```
static UINT8 matr2_Det( UINT8 A00, UINT8 A01, UINT8 A10, UINT8 A11 )
{
........
}

UINT8 main( UINT8 inp0_00, UINT8 inp0_01, UINT8 inp0_10, UINT8 inp0_11 )
{
........
   return matr2_Det(buf_00, buf_01, buf_10, buf_11);
}
```

The linker generates the following cell header (a schematic may be found in Fig. 10):

Example 43:

```
   ( library lib_main
      ( edifLevel 0 )
      ( technology ( numberDefinition ) )
      ( cell matr2_Det ( cellType GENERIC )
         ( view matr2_Det ( viewType NETLIST )
            ( interface
               ( port act ( direction input ) )
               ( port A00_0 ( direction input ) )
               ( port A00_1 ( direction input ) )
               ( port A00_2 ( direction input ) )
               ( port A00_3 ( direction input ) )
               ( port A00_4 ( direction input ) )
               ( port A00_5 ( direction input ) )
               ( port A00_6 ( direction input ) )
```

[14]The Electronic Design Interchange Format (EDIF) is a format used to exchange design data between different CAD systems, and between CAD systems and Printed Circuit fabrication and assembly.

```
                ( port A00_7 ( direction input ) )
                ( port A01_0 ( direction input ) )
                ( port A01_1 ( direction input ) )
                ( port A01_2 ( direction input ) )
                ( port A01_3 ( direction input ) )
                ( port A01_4 ( direction input ) )
                ( port A01_5 ( direction input ) )
                ( port A01_6 ( direction input ) )
                ( port A01_7 ( direction input ) )
                ( port A10_0 ( direction input ) )
                ( port A10_1 ( direction input ) )
                ( port A10_2 ( direction input ) )
                ( port A10_3 ( direction input ) )
                ( port A10_4 ( direction input ) )
                ( port A10_5 ( direction input ) )
                ( port A10_6 ( direction input ) )
                ( port A10_7 ( direction input ) )
                ( port A11_0 ( direction input ) )
                ( port A11_1 ( direction input ) )
                ( port A11_2 ( direction input ) )
                ( port A11_3 ( direction input ) )
                ( port A11_4 ( direction input ) )
                ( port A11_5 ( direction input ) )
                ( port A11_6 ( direction input ) )
                ( port A11_7 ( direction input ) )
                ( port CLK ( direction input ) )
                ( port GND ( direction input ) )
                ( port VCC ( direction input ) )
                ( port act_out ( direction output ) )
                ( port RFS_VAR_NAME_0 ( direction output ) )
                ( port RFS_VAR_NAME_1 ( direction output ) )
                ( port RFS_VAR_NAME_2 ( direction output ) )
                ( port RFS_VAR_NAME_3 ( direction output ) )
                ( port RFS_VAR_NAME_4 ( direction output ) )
                ( port RFS_VAR_NAME_5 ( direction output ) )
                ( port RFS_VAR_NAME_6 ( direction output ) )
                ( port RFS_VAR_NAME_7 ( direction output ) ) )
            ( contents
        ..................
    )
}
```

Each parameter generates 8 ports, since the bit size of the parameters is 8. The wires containing the return value is always called *RFS_VAR_NAME_....* The other ports are low-level steering and supply lines. Refer to section 3.10.3 for their meaning.

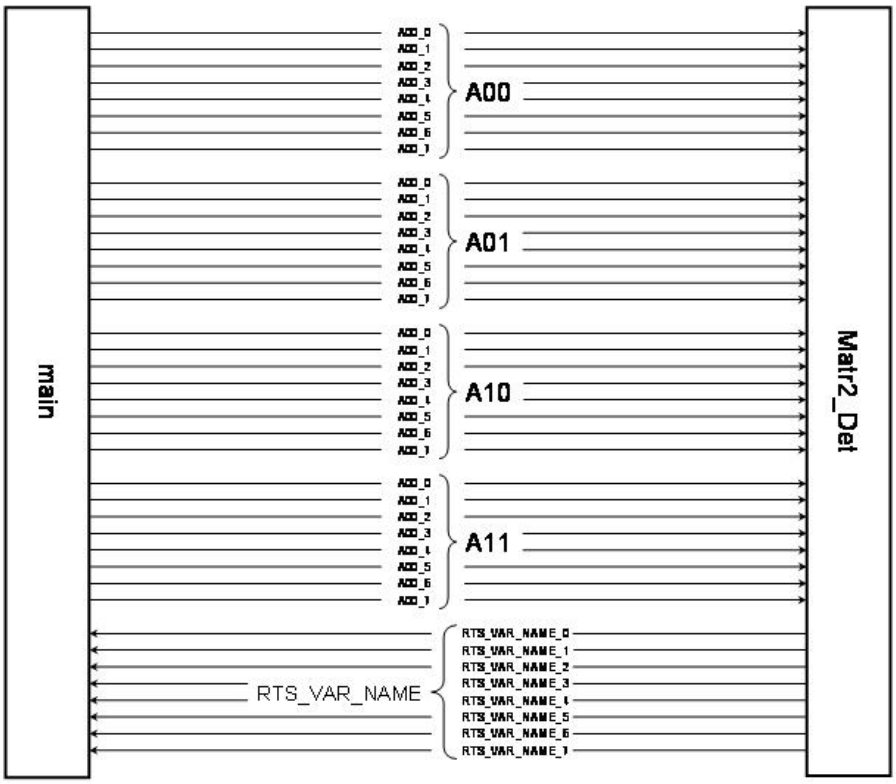


Figure 10: EDIF port bundle

3.10.3 Low-level signals

The circuit provides some signals for the basic interchange with the environment. They are common to all circuits and do not belong to a particular algorithm. Therefore the compiler or the linker generates them automatically. The user can focus his interests on the algorithm. Some of the signals are inputs to the circuit others are outputs. The low-level signals are listed below:

- *powersupply* (input)
- *ground* (input)
- *clock* (input)
- *activation* (input, high active)
- *readyforactivation* (output, high active)
- *outputready* (output, high active)

- *finished* (output, high active)

The first 3 signals are common for all kinds of clocked circuits. They are not influenced by the algorithm and therefore they are attached by the linker at the end of the compilation process. The ground and the power supply determine the basic logic levels 0 and 1, whereas the clock is the physical input clock and must not change during the operation time. In most cases it is connected directly to a clock source (i.e. a quartz). Clocks with different frequencies are generated internally if necessary. State-of-the-art FPGAs[15] or standard-cell-devices provide PLLs[16] to do this.

The last 4 signals are internal to our method. They are highly dependent on the particular algorithm except the activation signal. This signal starts the circuit. It must remain high at least for one clock cycle. But it may also remain high all the time. If the activation remains high it causes the circuit to run in an endless loop. If the activation remains high for less than one clock cycle the behaviour will be undefined, since it depends on the particular physical implementation of the circuit. The second internal signal "ready for activation" shows the environment of the circuit whether it is ready for activation or not. If the circuit is activated and the "ready for activation" signal shows busy state (logic low) it is ignored (therefore the activation signal is allowed to stay high all the time). Notice that the circuit may be ready for execution even if the entire circuit has not finished. This is typical for hardware implementations and show their pipeline character. The output ready signal shows that at least one output value is ready. Refer to section 5 for some examples showing the purpose of this signal. The last and least important signal is the "finished" signal. It stays active for one clock cycle. It may be later than the last value output for many reasons (i.e. update internal counters). But in most cases the information when the circuit has finished is not important to the circuits environment. It may be interesting for example in battery powered devices in order to avoid peaks in power consumption. The Fig. 11 shows a typical progress of the low-level-signals.

The time difference called ΔT_a in Fig. 11 determines the maximum input frequency and the time called ΔT_f is the total running time. Refer to section 4 for details on the timing of clocked circuits.

3.11 Edif generator

The output file format for the hardware target is EDIF[17]. The EIDF file is an ASCII file that contains all the information to reconstruct the silicon device. No additional file is necessary.

[15] Field Programmable Gate Array

[16] Short for phase-locked loop, an electronic circuit that controls an oscillator so that it maintains a constant phase angle on the frequency of an input, or reference, signal. A PLL ensures that a communication signal is locked on a specific frequency and can also be used to generate, modulate and demodulate a signal and divide a frequency.

[17] Electronic Data Interchange Format (refer to [17])

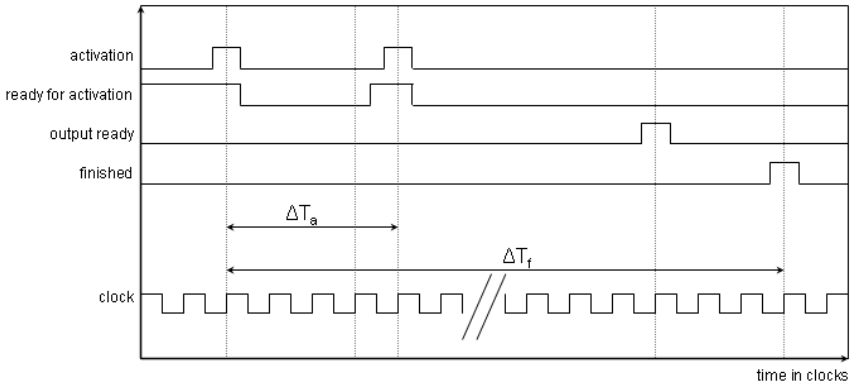


Figure 11: Progress of the low-level-signals

3.11.1 Generator Usage

The input files to the generator are the *.enl* files; refer to section 3.10.2. They contain our internal circuit descriptions, one each function. The calling convention for the EDIF generator is:

```
..../edif_gen [options] file.enl1 <file.enl2 file.enl3.... file.enl<n>>
options:
```

- $-t$ directory for the temporary files (default is the current directory)
- $-o$ EDIF output file name (the default uses the same base name as the first input file)
- $-a$ ATC list file name (list of the precompiled ATCs)
- $-e$ EDIF version (only version 2 is currently supported)
- $-l$ EDIF level (only level 0 (net list level) is currently supported)
- $-x$ vendor name of the target device (default is generic; currently supported are Altera and Xilinx devices)

The program works similar to a 'normal' linker. It reads all the input files and builds a module data base. Then it searches the *main* function) and processes recursively the design. If a new module is invoked the software goes through the data base and links the corresponding module. If no *main* function is found or a requested module is not found or any other inconsistency occurs than the EDIF generator will output an error with the necessary information to remove the error.

We have two kinds of modules. The ATC primitives and the modules generated from the source code. The main difference between these two module types is the way they are generated. The ATCs provide the basic functionalities like addition, multiplication etc. The ATCs are provided by the development suite. It is possible to change the code of the ATCs, since they are written

in C syntax and compiled by our compiler. As already told, the ATCs are written in a different way. The code is structural and not behavioural. Refer to VHDL documentation [16] for details on structural and behavioural descriptions. This method is also used in hardware description languages, but normally only the structural description is used in current design systems (like ISE[18], Quarturs2[19]).

The second type of modules are generated from the application software. Usually only these modules are written by the application engineer. The description type is behavioural. It has a higher level of abstraction than the structural one. 'Normal' C code is always behavioural. Refer to Fig. 12.

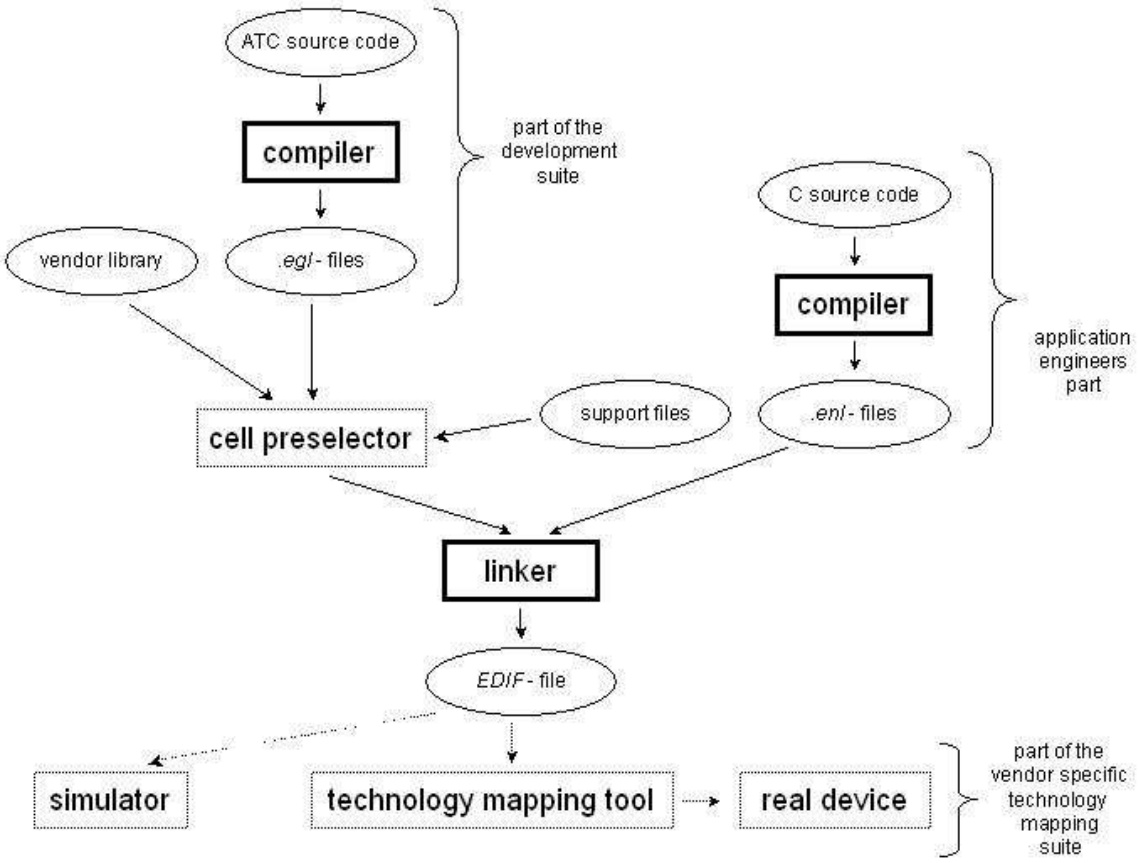


Figure 12: Linker Structure

Some modules may exist twice. Once within the vendor library and once within our internal library (*.egl*-files). In this case the user can choose one of the modules. The default is always our internal version. The listing of the desired vendor modules is done in a simple text file. Refer to section 3.11.2.

[18] the standard Xilinx design tool
[19] the standard Altera design tool

3.11.2 Support files

We use two support files to map our internal ATCs to EDIF cells. The first one with extension *.acm* maps the internal ATC to EDIF cell names. Refer to the following listing:

Example 44:

```
..................
+            I8A_c
-            I8S_c
*            I8M_c
/            I8D_c
>=           GreaterE8_c
!=           UnEqual8_c
==           Equal8_c
++           Inc8_c
--           Dec8_c
..................
}
```

For example the ATC that calculates the addition of two integer numbers (called *+*) is mapped to the cell called *I8A_c*. A cell with the name *I8A_c* must be implemented either in our library or in the vendor's library.

The second support file with extension *.dca* lists all the cells and their ports that are taken from the vendor's library (the default is always our internal library). Refer to the following listing:

Example 45:

```
..................
cell AND2     2 1 ports INPUT I0 INPUT I1 OUTPUT O
cell NAND2    2 1 ports INPUT I0 INPUT I1 OUTPUT O
cell OR2      2 1 ports INPUT I0 INPUT I1 OUTPUT O
cell OR3      3 1 ports INPUT I0 INPUT I1 INPUT I2 OUTPUT O
cell INV      1 1 ports INPUT I OUTPUT O
cell LD       2 1 ports INPUT G INPUT D OUTPUT Q
cell COMP8    2 1 ports INPUT A[8] INPUT B[8] OUTPUT EQ
cell FD       2 1 ports INPUT C INPUT D OUTPUT Q
..................
```

For example the 8bit comparator *COMP8* is taken from the vendor's library. Each line declares one cell. The common syntax is

```
cell <cell name> <numb. of inp. ports> <numb. of outp. ports> port <port list>
```

If a cell is taken from the vendor list it will be declared in the EDIF file without a body (it may be seen similar to an external declaration in a C file). The EDIF reader of the technology mapping tool (placing and routing tool) will add the body automatically.

4 Timing of Clocked Circuits

One of the most difficult tasks in hardware development is the net list generation and refinement. The net list is responsible for most of the characteristics of the final circuit, especially in terms of timing (i.e. clock rates, latencies etc) and gates consumption. At all times the tool developers have tried to find methods for totally automatic net list generation. Some of these methods allow circuit developers to use algorithmic languages, but unfortunately it is still necessary to have consolidated knowledge of the underlying hardware. Therefore people using these methods are hardware developers, whereas algorithm developers are rather software engineers. One can see that there is a gap in the chain: there is no easy way to implement systems described in an algorithmic language (i.e. a classical programming language).

It may be a good procedure to use a hardware description language to model the circuits. But there are a few disadvantages when using these tools: These languages are rather modelling languages than real programming languages, but algorithm and software designers tend to think sequentially and are used to dealing with programming languages like C or C++. Our approach is to use a programming language similar to C in order to enable software designers to develop synthesizable code. Pure C is not suitable, since it contains language constructs that have no immediate equivalent in hardware or at least are not synthesizable efficiently (i.e. pointers must be excluded). This thesis is focused on the timing behaviour of automatically synthesized circuits and their refinement.

Our approach to close the design gap is based on the methods published by Ian Page et. al. [1]. It utilizes a programming language, that is similar to C (called Handel-C) [1]. Hence we assume the reader to be familiar with ANSI C. The method published in [1] does not take care of the timing during compilation and offers no method for optimizing the generated net lists. Our method takes care of the timing behaviour during compilation and offers an effective method for partially automatic parallelization. The method has already been partially implemented.

The most significant trait of digital circuits is the grade of parallelization, since it determines the running time. Even some software based systems require concurrency, since parallel operating computers are available in great quantities. Therefore some of the considerations in this thesis are helpful for hardware as well as for software development.

The running time has a lower and an upper bound. The lower bound (ideal case) is equivalent to the running time of one elementary statement, since all the statements in this case run at the same time (maximum parallelism). Of course, this case is ideal and will not be achieved in real circuits. The upper bound is the worst case. In this case all the statements operate sequentially. The total running time is the sum of the running times of all the elementary statements (one pipeline system). The worst case can be considered as the baseline for the automatic parallelization.

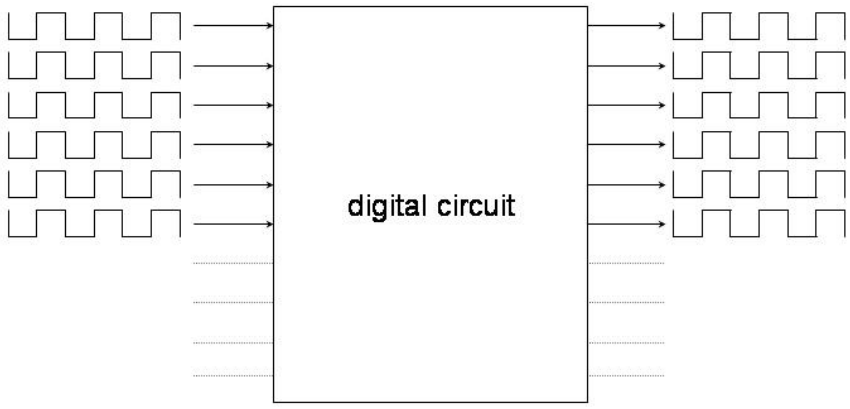


Figure 13: Delay of a design block

A second trait with a strong impact on the running time and the clock speed is the maximum latch time. If the processed data (or at least parts of it) have to be latched for a long time this will increase the running time and of course decrease the external clock speed (number of data sets operated in a particular time slot). The ideal case is no data latching at all. This means that each datum (even the intermediate data) is processed right after it has been generated. This, of course, is not achievable.

The method described below allows one to implement concurrency in two ways. First, the designer can embed parallel statements into the source code by using the keyword par, which has already been used by Handel-C. [3]. This is not the preferred method but sometimes cannot be avoided, especially within loops in the source code. Secondly, the compiler itself is able to distinguish between concurrent and sequential statements. If statements are marked as being concurrent, the compiler reorders them, otherwise they are operated sequentially. Anyway, the compiler is not able to find out all the concurrent statements. Therefore it is sometimes useful to be assisted by the designer.

4.1 Compilation process

The compilation is done in two steps by a smart compiler, which is capable of generating ARM assembler code as well as circuit net lists.

The first step is identical for both targets (assembler code or net lists). It generates an intermediate code. Each of these statements are sequential statements. Hence they are called atomic commands.

To ease the discussion we make some simplifying assumptions (simplifications). As will be shown later these do not constrain the usability of the presented method.

4.2 Assumptions

4.2.1 Functional units

The circuit is built of functional units, called atomic commands "ATC". They are assumed to be part of a library provided by the compiler vendor. The ATC is a logic operation unit. For example the addition of two integers is an atomic unit.

4.2.2 Data paths

The data paths are peer-to-peer connections between two ATCs. We do not admit bus accesses to a data pool. At first glance this assumption may cause restrictions. But it does not, for the following reason: The access to an external data storage (i.e. RAM block) has just to be added as another constraint during the net list generation. This does not cause a significant change because it does not influence the temporary succession. This of course, lowers the possibilities for parallelization. For example: Assume that there are two ATCs, which access the same external storage location and belong to different pipelines within a circuit. Without considering the bus access criterion the circuit synthesizer places them in order to operate at the same time. Using the additional criterion described above this cannot happen. The ATCs are placed into different time slots. The time slot is described in the next section (4.3.1).

4.2.3 Data manipulation

The data manipulations are disjoint in terms of time. That means, no connections are allowed between concurrent ATCs. This item expresses the fact that a particular datum can be manipulated just after the cycle within which it had been generated. Certainly it may be possible to allow connections between concurrent statements, especially if the sum of the execution times of both statements is lower than the specified overall execution time. But in this case an additional steering mechanism must be implemented (or at least a steering line between the two statements). Indeed this mechanism lowers the chance of automatic timing optimization.

4.2.4 Running time

We assume that all the ATCs have the same running time T_A. This is not really a restriction, since complex circuits with a long phyisical running time may be split and the parts connected by sequential lines. The outputs are generated at the latest at time T_A after receiving the inputs. If t_i is the time of the i-th cycle the outputs must not be generated later than $t = t_i + T_A$. We can always meet this requirement by setting T_A to the highest value. Hence if T_A^n is the running time of the ATC with index n and N is the number of ATCs then T_A will be calculated as $T_A = \max_{n \epsilon N} (T_A^n)$.

4.2.5 Connection length

There are no restrictions on the connection length. The only restrictions are given in 4.2.2 and 4.2.3. We assume that the compiler takes care of the synchronization of the data paths if necessary.

4.2.6 Data Transport

The data between ATCs are transported without time consumption. The consumption, which of course, is not zero in real implementations, can be ignored, since it is not comparable to the much higher switching time of the semiconductor gates. This assumption is used by many logic gates simulators.

4.2.7 Pipelining

The circuits support pipelining. New data samples will be processed even if the current sample is not finished. Refer to the sketch in Fig. 14.

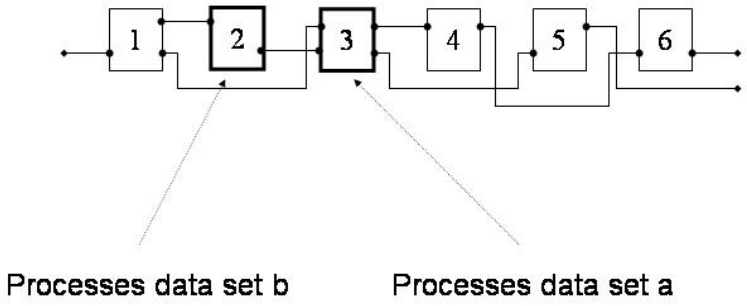


Figure 14: Pipelining

The results of each block are latched at their outputs. Therefore two consecutive blocks may process different data sets at the same time even if the result of block n is processed by the block $n + 1$ (i.e. block 3 processes the output of block 2).

4.2.8 IO cycles

The input/output cycles are equidistant in terms of time. The input data are read and the output data are generated in constant time slots, respectively. Equidistant input/output cycles are very typical for hardware devices. Example: We want to scan and classify items on an assembly line. If the assembly line runs with a constant velocity the camera will capture the images and forward them to the image processing engine in equidistant time steps.

4.3 Timing behaviour

There are several parameters characterizing the timing behaviour of digital circuits. We focus our interest onto the following two items:

- The total running time of the whole circuit (the time latency between the input and the output data)
- The maximal Input/output-frequency (the minimum time between two input data samples resp. output data)

4.3.1 Time slot matrix

One can imagine the circuit timing as a two dimensional matrix (called time slot matrix). Refer to Fig. 15.

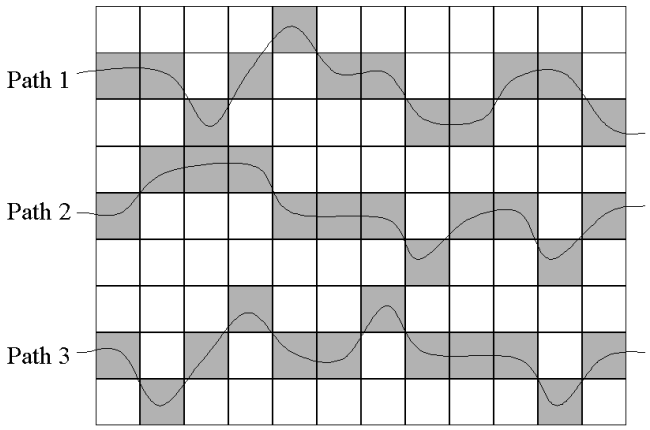


Figure 15: Time slot matrix

The rows represent the sequential ATCs whereas the columns represent the concurrent ATCs. Each of the matrix elements consumes the time T_A (refer to assumption 4.2.4). Starting at the time $t = 0$ on the left side of the matrix the time propagates to the right in equidistant steps (T_A). All the elements which belong to the same column are operated at the same time (but not all the time slots must necessarily be occupied). The used time slots are plotted with dark color in Fig. 15. The data is latched at the output of each ATC and passed to the next one each clock cycle. The data paths (pipelines) are indicated in the figure by lines.

4.3.2 Total running time

Calculation of the total running time is an easy task. Let $S = \{s^i\}$ be the set of all sequential data paths which connect an input to an output pin and $N(s^i)$ the Number of ATCs within

the particular path s^i. The total running time is calculated by

$$T_s = \max_i \left\{ N\left(s^i\right) \right\} * T_A \tag{2}$$

This means that the total running time is proportional to the number of ATCs within the longest data path (longest pipeline). Refer to Fig. 16. One can see that the longest data path in this example is the first one. The total running time in this example is 12 cycles. For synchronization it is necessary to latch the outputs. This may be done either on the output of the producing circuit or on the input of the consuming circuit. In our compiler we decided to latch the data on the outputs. This is just a convention and has no impact on the quality of the circuit. The shortest pipeline is the last one. Some of the ATC slots are not occupied. This means that the data have to be latched at the output of the last ATC until the clock activates the next one (next occupied ATC). If there are vacant matrix components at the beginning of a pipe line the data will be read at a later clock cycle. The data of the third pipeline in Fig. 16 is read at the third clock cycle.

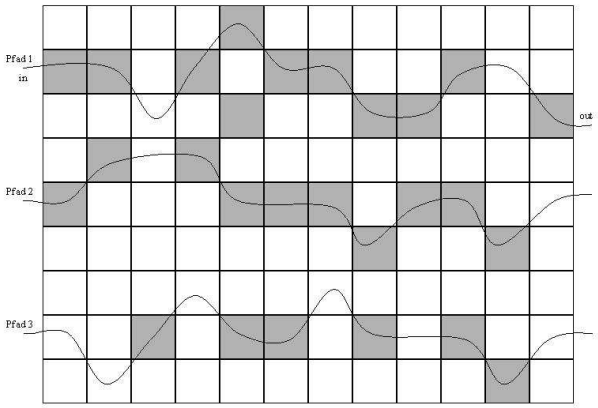


Figure 16: Time slot matrix 1

4.3.3 The maximal IO-frequency

The maximal IO-frequency depends on many parameters:

- Level of concurrency.
- The topology of the peer-to-peer connections. The longer the outputs are latched, the lower is the IO frequency (the number of input/output data sets per second).
- The running time of each ATC. Our assumption 4.2.4 sets this time to the constant $T_A^n = const = T_A$. In real implementations the running time of different types of ATCs is different. But the assumption 4.2.4 can be achieved formally by splitting "longer" ATCs

into a set of sequentially linked "shorter" ATCs. Therefore we can use assumption 4.2.4 for the formal calculations in section 4.3. But, since the subatomic ATCs that form the longer ATCs are linked sequentially, it is evident that their number has a strong impact on the IO-frequency. This time (T_A^n) depends on its type (i.e. adder, comparator...) and the physical implementation of the used basic gates. A lot of other parameters may influence the running time but they are not subject of discussion within the range of this thesis.

4.4 Formal ATC description

An ATC may be described by a vector. It is called ATC^n where n is the index $1 \leq n \leq N$. N is the total number of ATCs within a circuit. Each ATC^n is represented by a 5-tuple:

$$ATC^n = \begin{pmatrix} type \\ t_i \\ i_j \\ INPUT\left(IO-type, ATC^{\alpha}, ..., ATC^{\beta}\right) \\ OUTPUT\left(IO-type, ATC^{\gamma}, ..., ATC^{\delta}\right) \end{pmatrix}$$

The set of all the ATCs is defined by $ATC^N = \{ATC^n\}$ where N is the total number of ATCs: $1 \leq n \leq N$.

- *type*: Type of the ATC ((i.e. adder, comparator...)). The type is given just for sake of completeness.
- t_i: Horizontal index within the time slot matrix (time when the ATC is processed)
- i_j: Vertical index within the time slot matrix. The vertical index does not influence the timing. It is just given to mark the time slot as used.
- $INPUT(...)$: Input connections; the components ATC^{γ} are the ATCs which are connected to the inputs; $IO-type$ is the connections type (i.e. the Bit width)
- $OUTPUT(...)$: Output connections; the components ATC^{γ} and $IO-type$ are similar to the INPUTs.

4.5 Optimizing the timing behaviour

The basic idea for the timing optimization is a smart reordering of the components of the time slot matrix. It is evident that shifting the components to the left yields lower circuit running times. But a reordering may have negative side effects; i.e. the outputs must be latched for a longer time. Our approach to this optimization problem is the usage of the well known dynamic programming method. Reordering is allowed just if it minimizes the cost function defined below. Hence the main task is to find a cost function, that has minimal value for the best time slot configuration, or at least has a minimum value for a configuration, that is very close to the optimum.

4.5.1 Cost function

In this chapter we introduce the cost function F used to solve the optimization problem verbalized in the last section. The cost function assigns a real number to each possible configuration of the time slot matrix. We focus our interest onto the following questions:

- Does the function tend to a stable value when propagating in time or is it oscillating around a stable value?
- Is there more than one minimum? How can we choose the right one (distinguish between local and absolute minima)? How can we define further criteria in this case?
- How does the function depend on its parameters? (Definition of the cost function).

The cost function F evidently depends on the time slot configuration. This configuration changes during optimization. Therefore it also depends indirectly on time, where time is understood as the number of reconfigurations of the time slot matrix, since the optimization started. Formally, we denote the function by

$$F = F\left(ATC^N(t)\right)\ t\epsilon\mathcal{N} \tag{3}$$

- ATC^N: the compilation of all the vectors ATC^n defined in the last paragraph
- t: the index number of the optimization step (the start configuration is assigned the index 0 and is incremented after each optimization step)

$F\left(ATC^N(t)\right)$ is a function which measures the quality of a particular time slot configuration. The lower the function value, the better is the circuit timing.

One of the most important constraints for the cost function is expressed in the following equation:

$$\text{if } F\left(ATC_1^N\right) = F\left(ATC_2^N\right) \text{ then } ATC_1^N = ATC_2^N \tag{4}$$

The equation above may be interpreted as follows: If two final ATC configurations (called ATC_1^N and ATC_2^N) have the same cost function, they are identical (when we talk about final ATCs, we meen ATCs with minimal cost function). Line exchanges of the ATC matrix define a unitary transformation and are therefore called identical. And of course, they have the same cost function.

As already mentioned in the last paragraph there are some additional constraints to be considered:

- Only unused time slots may be occupied. If $ATC^n = (\cdots t_i, i_j \cdots)$ and $ATC^m = (\cdots t_k, i_l \cdots)$ are two ATCs in the same circuit then t_i must not be identical with t_k or i_j must not be identical with i_l ($t_i \neq t_k$ or $i_j \neq i_l$).

- Before processing an ATC, the calculation of all its inputs has to be finished $\Longleftrightarrow$ If $ATC^n = (\cdots, t_i, \cdots, INPUT\,(\cdots ATC^\gamma\,(\cdots, t_k, \cdots)))$ then $t_i > t_k$.
- The same data must not be manipulated within concurrent ATCs. Let ATC^m and ATC^n be located in concurrent data paths. At the source code level this restriction is equivalent to the following statement.

  Example 46:

```
{
   int x;

   par
   {
      x = 3;
      x = 4;
   }
}
```

  The variable x is manipulated at the same time. This, of course, is a faulty program.

  If $ATC^n = (\cdots, OUTPUT\,(\cdots, ATC^\gamma, \cdots))$ and
  $ATC^m = \left(\cdots, OUTPUT\left(\cdots, ATC^\delta, \cdots\right)\right)$ then $\gamma \neq \delta$.

During the first step we don't take care of the propagation. For this reason we can ignore the parameter t: $F\left(ATC^N\,(t)\right) = F\left(ATC^N\right)$. We define the cost function as a sum over the number of ATCs:

$$F\left(ATC^N\right) = \sum_{n=1}^{N} F\,(ATC^n) + F_0 \tag{5}$$

From assumption [4.2.4] it follows that the running time does not depend on the type of the ATC. In fact only the horizontal position within the time slot matrix and the length of the IO-lines contribute to the cost function. Exchanging the positions of two concurrent ATCs or moving an ATC vertically has no effect on the cost function. Therefore we can remove the ATC type from its parameter list:

$$F\,(ATC^n\,(type, \cdots)) = F\,(ATC^n\,(\cdots)) \tag{6}$$

The parameters $INPUT\left(ATC^\alpha, ATC^\beta, \cdots\right)$ and $OUTPUT\left(ATC^\alpha, ATC^\beta, \cdots\right)$ specify connections for the data input and data output respectively. The union set of the inputs and outputs is the same:

$$\bigcup_{n=1}^{N} INPUT\,(ATC^\alpha, \cdots) = \bigcup_{n=1}^{N} OUTPUT\,(ATC^\alpha, \cdots) \tag{7}$$

On that account it is sufficient to consider either the inputs or the outputs. Hence we restrict ourselves to the inputs. This yields the following dependence:

$$F\left(ATC^n\right) = F\left(t_i, i_j, INPUT\left(ATC^\alpha, ATC^\beta, \cdots\right)\right) \tag{8}$$

The contribution of each ATC to the cost function depends on its position within the time slot matrix t_i, i_j and its input connections $INPUT\left(ATC^\alpha, ATC^\beta, \cdots\right)$.

We use the following approach for $F(ATC^n)$:

- The function F depends linearly on the sequential position t_i.

$$F\left(ATC^n\right) \sim t_i \tag{9}$$

- A quadratic contribution is added due to input connection length.

$$F\left(ATC^n\right) \sim \max_{i \neq j} \left(t_i - t_j\right)^2 \tag{10}$$

if ATC^n is connected to an ATC located in row t_j of the time slot matrix. We use a quadratic dependence for the wire length, since in real silicon the routing of very long wires is very difficult. Long wires also reduce the maximum data input frequency, since the data has to be latched for a long time. Therefore we gain in total running speed if we can avoid very long wires. Refer to Fig. 17. We examine the cost function for the ATC

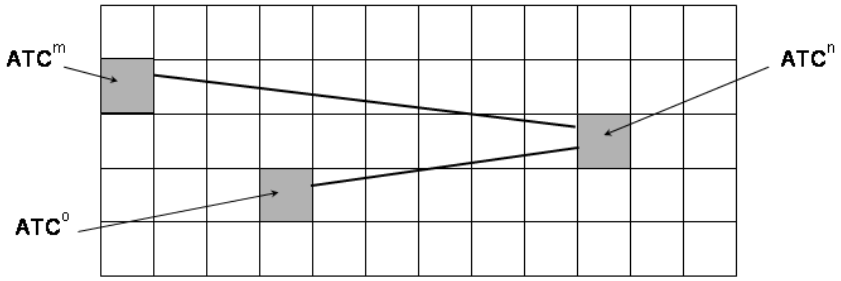


Figure 17: Wire length term of the cost function

with index n (ATC^n). We assume that this ATC has two inputs. The data are generated by ATC^m and ATC^o. Only the longest link has to be considered (in our sketch the link between ATC^m and ATC^n). The column index of ATC^m is 0 whereas the column index of ATC^n is 9. The difference is 9. The cost function calculator will include the following factor in the cost function of ATC^n:

$$\max_{i \neq j} \left(t_i - t_j\right)^2 = 9^2 = 81 \tag{11}$$

Example: Consider the following time slot configuration (Fig. 18) as a starting point for the optimization. At the beginning all the functional modules (ATCs) are ordered

sequentially. One can see that this configuration takes 6 cycles to generate its outputs, assuming that each of the modules takes exactly one cycle (assumption 4.2.4).

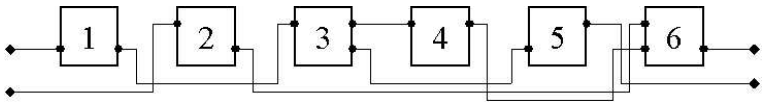


Figure 18: ATC optimization; starting point

Without the contribution due to input connection length the first optimisation step yields (Fig. 19):

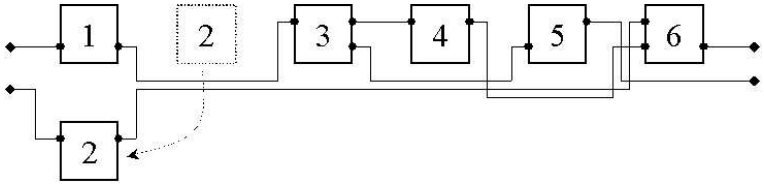


Figure 19: optimized ATC; without consideration of the connection length

The input connections contribution increases the cost function. Therefore this configuration change will not be done. Instead the following will happen (Fig. 20):

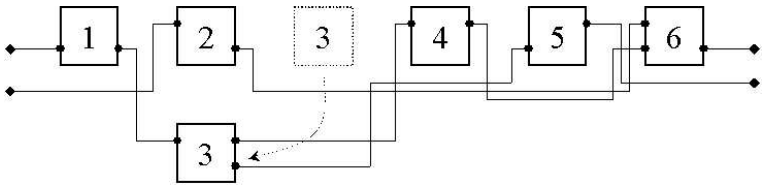


Figure 20: optimized ATC; with consideration of the connection length

Find below (Fig. 21) the final configuration with and without the input connection costs (on the left hand respectively on the right hand).

The output of the ATC^2 on the left hand side has to latch its outputs for 3 cycles. It is free to process new data just after ATC^6 has finished its calculations. On the right hand side the latch time of ATC^2 is only 2 cycles. The second one allows a higher external clock speed, since the outputs of ATC^2 is latched for a shorter time.

Summation of the different cost function components yields the following function for each ATC:

$$F(ATC^n) = A * t_i + B * \max_{i \neq j} (t_i - t_j)^2 \quad \text{if}$$
$$ATC^n = (t_i, i_j, INPUT\,(t_j)) \tag{12}$$

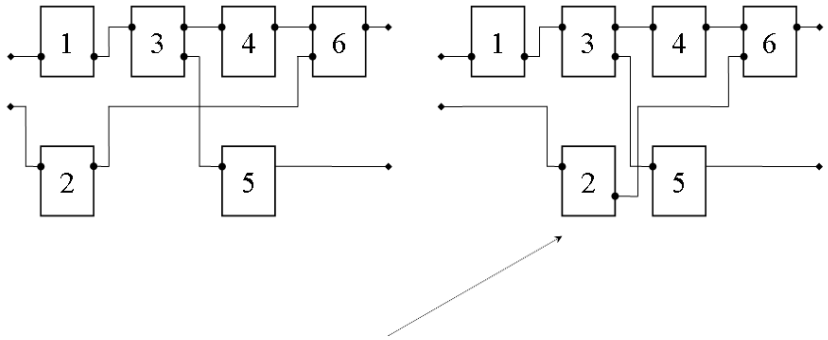


Figure 21: optimized ATC; with and without consideration of the connection length

Summation over the number of ATCs:

$$F(ATC_1^N) = \sum_{n=1}^{N} \left(A * t_i^n + B * \max_{i \neq j} \left(t_i^n - t_j^m \right)^2 \right) \tag{13}$$

$$\Rightarrow F(ATC_1^N) = \left(A * \sum_{i=1}^{i^{max}} \sum_{j=1}^{j^{max}} t_i + B * \max_{i \neq j} \left(t_i^n - t_j^m \right)^2 \right)$$

5 Experimental results

We will demonstrate in this section the mode of operation of the compiler, considering as examples the matrix multiplication and the determinant calculation. The program multiplies two matrices and calculates the determinant of the result. The calculation is demonstrated with and without optimization.

Refer to appendix 9.2 for the source code of the matrix multiplication. We discuss in this section two milestones of the compilation process:

- The *intermediate* code
- The final silicon design and the ARM *assembler* code (the assembler code is trivial for experienced assembler programmer; it is mentioned only for completeness)

5.1 Compiler inputs and outputs

Please refer to section 3.7.1 for a detailed list of the command line options and the default settings. The simplest shell command using the default settings is:

Example 47:

```
>cchw matr2Mult.hc
```

A more usual program call will use the optimizer. The shell command in this case is:

Example 48:

```
>cchw -edf -x Xilinx -O2 -trg nl matr2Mult.hc
```

We've used the hardware target (*-trg nl*). This selection is arbitrary in this section, since the intermediate code is identical for both targets. An EDIF file will be generated (*-edf*) and the selected target device vendor is Xilinx (*-x Xilinx*). The most important compiler switch is *-O2*. It selects the optimizer. The optimizer stage *2* is for future use.

The compiler's response is the following:

Example 49:

```
Net list generation
===================
Found 4 functional blocks.

Optimization
============
```

```
matr2_Det                  7 iterations
dummy                      2 iterations
matr2Mult_1elem            7 iterations
main                       823 iterations

Timing
======
Global clock frequency is 300 MHz
Longest path = 6 cycles (max. input frequency 50 MHz)
Circuit delay = 12 cycles (40.00 nanoseconds)

Edif file generation
====================
Link 3 user defined functional blocks.
Link 13 primitives from the ATC library.
Link 5 primitives from the Xilinx library.
```

The compiler messages are split into 4 parts:

- The net list *generator* (synthesizer)
- The net *optimizer*
- The timing *analizer*
- The edif file *generator* (linker)

The source code used in this example contains 4 functional blocks (functions) but only 3 are used (one dummy function was inserted for demonstration purposes). 13 additional modules are linked in from our internal library and 5 from the Xilinx library.

The optimizer messages give us the number of iterations for each module (a module corresponds to a function in the source code).

The most interesting parts are the timing messages. They contain the main timing information: the circuit delay and the longest path. Refer to section 4 for details on the circuit timing. The circuit delay in our example is 12 cycles (corresponds to 40 nanoseconds if a 300 MHz clock is used). That means that the output values are generated 40 nanoseconds after the circuit was activated. The longest path in our example takes 6 cycles to forward the data from the start to the end point. That means that the circuit may be reactivated every 6 cycles even if it has not been finished at this time.

5.2 Intermediate code

As already mentioned, the intermediate code is identical for both targets (hardware and software). We start without intermediate code optimization. Please find below the intermediate code for our sample application. Please notice that the index listed in the first row does not determine the real time slot index. It is only a sorting index.

Example 50:

```
matr2_Det
=========
(0)  (TS) 1 <--|     A00 <--|    A01 <--|    A10 <--|    A11 <--|
(1)  (TS) 2 <-- [1]  (*) CCHW_TMPVAR_18 <-- [A10 A01]
(2)  (TS) 3 <-- [2]  (*) CCHW_TMPVAR_17 <-- [A00 A11]
(3)  (TS) 4 <-- [3]  (-) buf1 <-- [CCHW_TMPVAR_17 CCHW_TMPVAR_18]
(4)  (TS) 5 <-- [4]  (+) buf1 <-- [buf1 A11]
(5)  (TS) 6 <-- [5]  (=) buf2 <-- [buf1]
(6)  (TS) 7 <-- [6]  (=) RFS_VAR_NAME <-- [buf2]
(7)  (TS) |<-- [7]   (RFS) [RFS_VAR_NAME]


dummy
=====
(0)  (TS) 1 <--|     A00 <--|    A01 <--|
(1)  (TS) 2 <-- [1]  (*) buf <-- [A00 A01]
(2)  (TS) 3 <-- [2]  (=) RFS_VAR_NAME <-- [buf]
(3)  (TS) |<-- [3]   (RFS)


matr2Mult_1elem
===============
(0)  (TS) 1 <--|     A0 <--|    A1 <--|    B0 <--|    B1 <--|
(1)  (TS) 2 <-- [1]  (*) CCHW_TMPVAR_31 <-- [A1 B1]
(2)  (TS) 3 <-- [2]  (*) CCHW_TMPVAR_30 <-- [A0 B0]
(3)  (TS) 4 <-- [3]  (+) buf <-- [CCHW_TMPVAR_30 CCHW_TMPVAR_31]
(4)  (TS) 5 <-- [4]  (=) RFS_VAR_NAME <-- [buf]
(5)  (TS) |<-- [5]   (RFS) [RFS_VAR_NAME]


main
====
(0)  TS) 1 <--|      inp0_00 <--|   inp0_01 <--|   inp0_10 <--|   inp0_11 <--|
```

```
                    inp1_00 <--|   inp1_11 <--|   inp1_01 <--|   inp1_10 <--|
(1)  TS) 2 <-- [1]   (=) inp1Buf_00 <-- [inp1_00] (=) inp1Buf_11 <-- [inp1_11]
(2)  TS) 3 <-- [2]   (=) inp1Buf_01 <-- [inp1_01] (=) inp1Buf_10 <-- [inp1_10]
(3)  TS) 4 <-- [3]   (BS) buf_00 <-- [(inp0_00,inp0_01,inp1Buf_00,inp1Buf_10)]
(4)  TS) 5 <-- [4]   (BS) buf_01 <-- [(inp0_00,inp0_01,inp1Buf_01,inp1Buf_11)]
(5)  TS) 6 <-- [5]   (BS) buf_10 <-- [(inp0_10,inp0_11,inp1Buf_00,inp1Buf_10)]
(6)  TS) 7 <-- [6]   (BS) buf_11 <-- [(inp0_10,inp0_11,inp1Buf_01,inp1Buf_11)]
(7)  TS) 8 <-- [7]   (=) out_00 <-- [buf_00]
(8)  TS) 9 <-- [8]   (=) out_01 <-- [buf_01]
(9)  TS) 10 <-- [9]  (=) out_10 <-- [buf_10]
(10) TS) 11 <-- [10] (=) out_11 <-- [buf_11]
(11) TS) 12 <-- [11] (BS) CCHW_TMPVAR_49 <-- [(buf_00,buf_01,buf_10,buf_11)]
(12) TS) 13 <-- [12] (=) RFS_VAR_NAME <-- [CCHW_TMPVAR_49]
(13) TS) |<-- [13]   (RFS) [RFS_VAR_NAME]   [out_00] |-->   [out_01] |-->
                    [out_10] |-->    [out_11] |-->
```

The example code has four modules (one of them (*dummy*) is not used; since it is not invoked by any other function, the linker ignores it). Each one has its corresponding function in the source code. We take a look at the first module called *matr2_Det*. It calculates the determinant of a $2x2$-matrix. The first line (with index *0*) contains the four input values (components of the matrix). They are read concurrently, since the input values are uncorrelated even if the optimizer is not activated. The second and the third commands (indices *2* and *3*) calculate the two multiplications of the diagonal components. These two multiplications may be calculated at the same time but in this example we did not switch on the optimizer. The results are stored in the temporary variables *CCHW_TMPVAR_15* and *CCHW_TMPVAR_16*. The return variable is *buf*. In this case the return value is stored in a variable defined in the source code. But the return value may also be stored in a temporary variable. The last command is always *RFS* followed by a return value or not. If a return value is required it is always copied to a temporary variable called *RFS_VAR_NAME*. A *C* function may have two or more conditional *return* statements. In this case all these values are copied to *RFS_VAR_NAME*. Refer to the following example:

Example 51:

```
static UINT8 dummy( UINT8 A0, UINT8 A1 )
{
   UINT8 buf0, buf1;


   buf0 = (A1-A0);
   buf1 = (A0-A1);
```

```
   if (buf0 > buf1)
      return buf0;
   if (buf0 < buf1)
      return buf1;
}
```

The intermediate code for this short dummy example is listed below.

Example 52:

```
dummy
=====
(0)     (TS) 1 <--|             A0 <--|          A1 <--|
(1)     (TS) 2 <-- [1]          (-) buf0 <-- [A1 A0]
(2)     (TS) 3 <-- [2]          (-) buf1 <-- [A0 A1]
(3)     (TS) 4 <-- [3]          (>) CCHW_TMPVAR_27 <-- [buf0 buf1]
(4)     (TS) 5 <-- [4]          (if) [CCHW_TMPVAR_27 6]
(5)     (TS) 6 <-- [5]          (=) RFS_VAR_NAME <-- [buf0]
(6)     (TS) 7 <-- [6]          (<) CCHW_TMPVAR_29 <-- [buf0 buf1]
(7)     (TS) 8 <-- [7]          (if) [CCHW_TMPVAR_29 9]
(8)     (TS) 9 <-- [8]          (=) RFS_VAR_NAME <-- [buf1]
(9)     (TS) |<-- [9]           (RFS) [RFS_VAR_NAME]
```

Depending on the values of variable *buf0* and *buf1* either the value of *buf0* is returned or the value of *buf1*. Formally both values are copied to the temporary return variable *RFS_VAR_NAME*.

The *RFS* statement of the *main* function has a special feature. The common return variable *RFS_VAR_NAME* of the main function in example 50 is followed by four additional output values *out_00*, *out_01*, *out_10* and *out_11*. These variables are defined as outputs; refer to section 2.1.3. Within the intermediate code they are always part of the *RFS* statement of the *main* function. The input variables are always part of the input statement of the *main* function (the first line after the function name with index *0*).

5.3 Software target

As already mentioned, the C compilation process for software targets is nowadays a well known task. Normally each assembler statement may be assigned a source code line (this may not be the case in some highly optimizing compilers). This feature is used by debuggers to highlight the currently active source code line in a debug session. The users of our compiler can use the *-i* option to interleave the C source code lines as comments and the assembler code in the assembler file; refer to section 3. Take a look at the code snippet in example 53. The additional assembler statements (i.e. *STMDB SP!,R1-R3,LR*) are part of the function invocation process and inserted automatically by the code generator. These parts of the assembler code may be

much more complicated for example if parameters are pushed onto the stack. Our example is simple, since the parameters are already stored in the right registers.

Example 53:

```
;static UINT8 matr2_Det
;  (
;    UINT8 A00, UINT8 A01,
;    UINT8 A10, UINT8 A11
;  )
matr2_Det PROC
        STMDB   SP!,{R1-R3,LR}          ;Save the used registers

;   buf1 = (A00*A11) - (A10*A01);
        MUL     R1,R2,R1
        MUL     R0,R3,R0
        SUB     R0,R0,R1
;   return buf1;
|LABEL.matr2_Det.4|

        LDMIA   SP!,{R1-R3,LR}          ;Restore the used registers
        BX      LR              ;Return from subroutine
        ENDP
```

The code snippet above is the assembler code for the determinant calculation. The source code has two statements, the calculation itself (two multiplications and one addition) and the return statement. These source code lines are interleaved as comments (lines starting with the comment delimiter *;*) followed by the corresponding assembler statements.

5.4 Hardware target

We start the discussion without optimization. The compiler generates a circuit that processes the algorithm almost fully sequentially. Only the input and output channels are activated concurrently. Refer to the sketches in Fig. 22 (Figures 22, 24, 25 and 26 are screen shots).

The first sketch is the determinant calculation and the second one the matrix multiplication (we did not print the schematic of the main routine, since it does not fit on the screen).

Now we switch on the optimizer. One can see that the circuit delay decreases, since some circuits are processed concurrently if possible. Refer to the sketches in Fig. 24, Fig. 25 and Fig. 26.

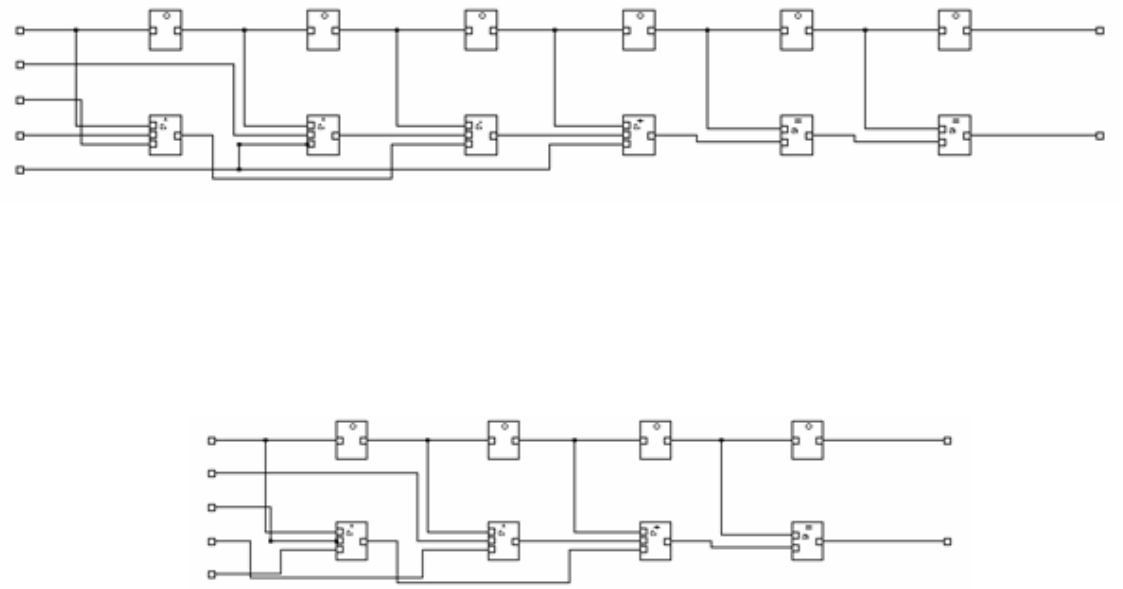


Figure 22: Determinant calculation and matrix multiplication without optimization

5.4.1 Simulation

The model simulation is the easiest way to verify the generated design. We have used a digital simulator called *ModelSim*[20] from *Mentor Graphics*. The simulator is fed with two input files:

- The design file contains the generate design.
- The test bench contains the input parameters and the timing constraints.

The simulator calculates the logic levels of the signals as a function of time and draws them to the *wave* window. A snapshot of the wave window is given in Fig. 23. It shows the signals over a period of 200 nanoseconds. Only the input and output signals are given. In our simulation the circuit was activated after 100 nanoseconds (see the signal with label *act*). The calculation is finished 80 nanoseconds after its activation (see signal with label *act_out*). The input clock frequency was 100MHz (see the signal with label *clock*).

[20]ModelSim XE III/Starter 6.0d, Revision: 2005.04, Date: Apr 26 2005

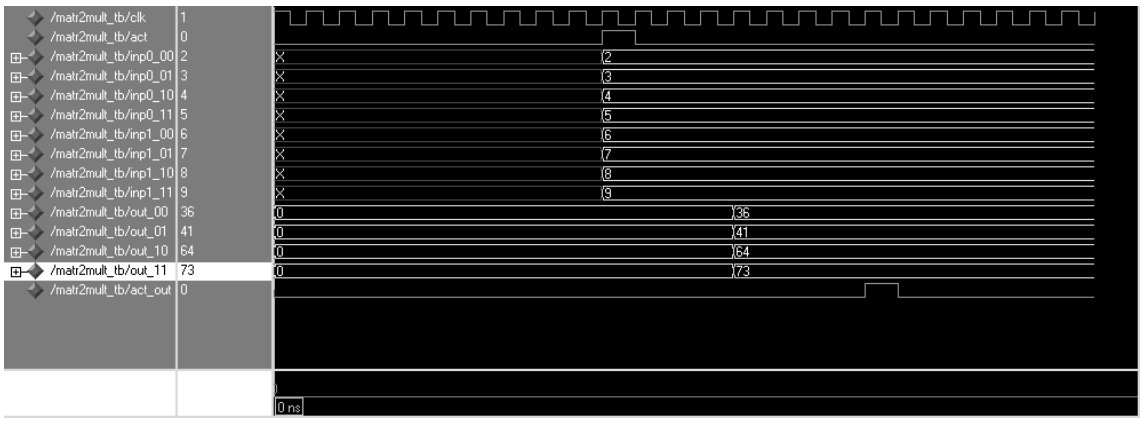


Figure 23: Simulation of the matrix multiplication

One can see, that the calculated values are correct and the circuit delay corresponds to the number of operations needed for a 2x2 Matrix Multiplication. The circuit terminates later (refer to signal

`/matr_mult_tb/act_out`

) because the determinant of the generated matrix is also calculated (but not displayed in the wave window).

Please refer to appendix 9.4 for the test bench used.

5.5 Timing constraints

In this section we show the usage of timing constraints within accesses to input/output channels. Without additional timing constraints the access to the channels is done immediately after the circuit is activated. Sometimes it is necessary to read or write the data of different variables via the same channel. In this case the data for different variables is provided by the external circuit at different times (time slicing). Therefore the data must be read and stored in the corresponding variable at the right time. This situation is given for example if the data is read or written via an external bus. Refer to the following example:

Example 54:

```
input inp1;

UINT8 matr2Mult
  (
     UINT8 inp0_00,
     UINT8 inp0_01,
     UINT8 inp0_10,
     UINT8 inp0_11
```

```
  )
{
   UINT8 buf_00;
   UINT8 buf_01;
   UINT8 buf_10;
   UINT8 buf_11;

   UINT8 inp1Buf_00, inp1Buf_01, inp1Buf_10, inp1Buf_11;
   inp1Buf_00 = inp1;
   inp1Buf_01 = inp1;
   inp1Buf_10 = inp1;
   inp1Buf_11 = inp1;

   buf_00 = matr2Mult_1elem(inp0_00, inp0_01, inp1Buf_00, inp1Buf_10);
   buf_01 = matr2Mult_1elem(inp0_00, inp0_01, inp1Buf_01, inp1Buf_11);
   buf_10 = matr2Mult_1elem(inp0_10, inp0_11, inp1Buf_00, inp1Buf_10);
   buf_11 = matr2Mult_1elem(inp0_10, inp0_11, inp1Buf_01, inp1Buf_11);

   out_00 = buf_00;
   out_01 = buf_01;
   out_10 = buf_10;
   out_11 = buf_11;

   return matr2_Det(buf_00, buf_01, buf_10, buf_11);
}
```

The circuit in example 54 has the same core functionality as the example in appendix 9.2; it calculates the product of two matrices. The only difference is that the components of the second matrix are read via the same channel (represented by the external variable *inp1*). The code in example 54 works only if the four input statements are executed sequentially. This is the case only if the optimizer is switched off and the four statements are not embedded in a *par*-block. Otherwise the four values are read at the same time. This, of course, is not possible via the same channel.

The easiest way to avoid this problem is shown in example 55. We assume that the external circuit provides the components sequentially at $t_{c_0} = 0$ clock cycles, $t_{c_1} = 1$ clock cycles, $t_{c_2} = 2$ clock cycles and $t_{c_3} = 3$ clock cycles.

Example 55:

```
.....................
   inp1Buf_00 = inp1 <0 clocks>;
```

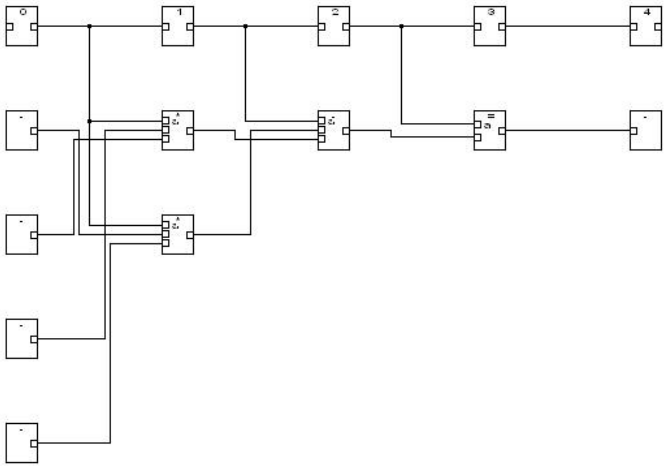


Figure 24: Determinant calculation

```
   inp1Buf_01 = inp1 <1 clocks>;
   inp1Buf_10 = inp1 <2 clocks>;
   inp1Buf_11 = inp1 <3 clocks>;
.....................
```

In this case we specify the clock index. The specified clock cycles must be different for each of the statements. Please notice that in this case it makes no difference if the statements are embedded in a *par*-block or not.

The second option that has no unique timing constraints is shown in example 56.

Example 56:

```
.....................
   inp1Buf_00 = inp1 <0>;
   inp1Buf_01 = inp1 <3 nanoseconds>;
   inp1Buf_10 = inp1 <6 nanoseconds>;
   inp1Buf_11 = inp1 <9 nanoseconds>;
.....................
```

In this case we use nanoseconds, seconds etc. This specification is not unique, since the compiler has to round the timing constraints to the finest time step which is one clock cycle. The timing constraint in example 57 has the same timing behaviour as in example 56 even though they are not identical. The timing delay in both examples is one clock cycle, since the cycle duration is 3.33 nanoseconds if an external clock of 300 MHz is assumed.

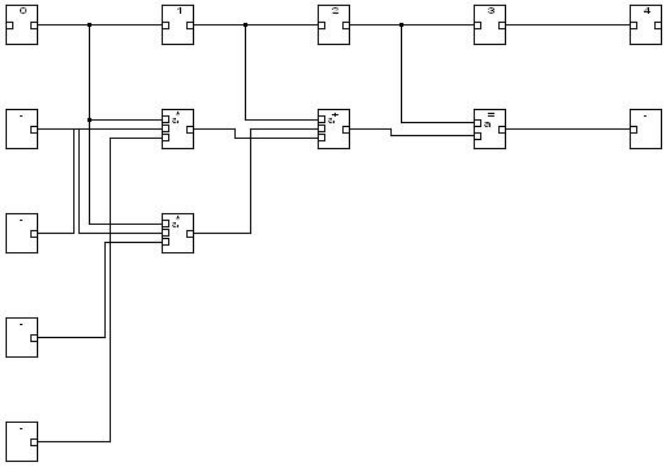


Figure 25: Matrix multiplication

Example 57:

```
.....................
   inp1Buf_00 = inp1 <0>;
   inp1Buf_01 = inp1 <3 nanoseconds>;
   inp1Buf_10 = inp1 <7 nanoseconds>;
   inp1Buf_11 = inp1 <10 nanoseconds>;
.....................
```

It is evident, that the relative timing behaviour does also depend on the input clock frequency if nanoseconds, microseconds etc are used.

All the examples above generate the same circuit if the frequency is fixed to 300 MHz; If the frequency is much lower than that the compiler will insert wait states into the second and the third circuits between the inputs of the different values.

5.5.1 Inappropriate timing constraints

In most cases one will use the optimizer in order to get an optimal circuit. In this case the user does not know anything about the timing behaviour of the final circuit. It may be possible that useless constraints are specified. If input constraints only are used the situation will be quite easy. The user has only to take care that the same input channel is not accessed twice at the same time. The compiler prints a warning in this case. It does not print an error, since sometimes it is desired to assign two different variables the same input value.

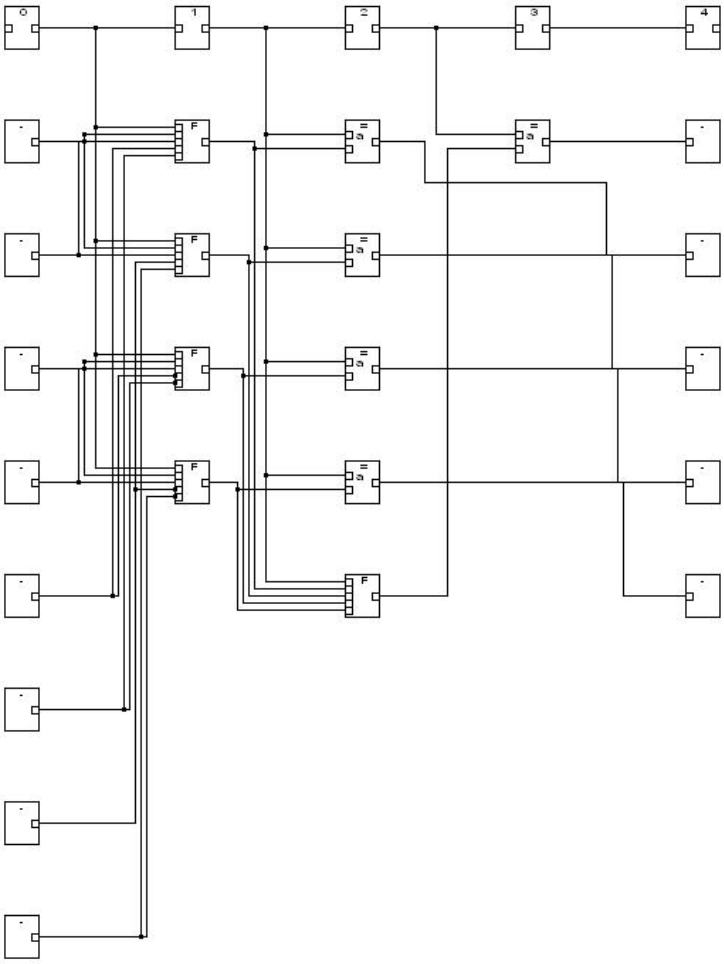


Figure 26: Main routine

The situation is more difficult if output timing constraints are used; and the most difficult situation arises if input and output timing constraints are used. The example in Appendix 9.2 takes 12 clock cycles for execution.

If the user specifies that one of the outputs must be written after less than 12 clock cycles, it will not be possible, since the value has not been generated at this time. This situation causes the compiler to generate an error. The user is forced to change his constraints.

5.5.2 Relative output timing constraints

The timing constraints are always given relatively to the circuit activations. This is useful for input channels in most cases, since reading the input values is usually the first thing to do. The situation is different for output channels. In many cases it is not important to write the output values after an exact number of clock cycles, but in an exact relation to each other. This is the case when using the same channel for more than one variable (i.e. an external data bus is used). Therefore the timing constraints may also be given relative to the output ready signal

(refer to section 3.10.3 for details on ready/busy signals).

Refer to example 58.

Example 58:

```
OUTPUT UINT8 out;
INPUT UINT8 inp1;
.....................
UINT8 main
  (
     UINT8 inp0_00,
     UINT8 inp0_01,
     UINT8 inp0_10,
     UINT8 inp0_11
  )
.....................
   buf_00 = matr2Mult_1elem(inp0_00, inp0_01, inp1Buf_00, inp1Buf_10);
   buf_01 = matr2Mult_1elem(inp0_00, inp0_01, inp1Buf_01, inp1Buf_11);
   buf_10 = matr2Mult_1elem(inp0_10, inp0_11, inp1Buf_00, inp1Buf_10);
   buf_11 = matr2Mult_1elem(inp0_10, inp0_11, inp1Buf_01, inp1Buf_11);

   out = buf_00  <0 clocks relativeOut>;
   out = buf_01  <1 clocks relativeOut>;
   out = buf_10  <2 clocks relativeOut>;
   out = buf_11  <3 clocks relativeOut>;

   return matr2_Det(buf_00, buf_01, buf_10, buf_11);
.....................
```

In this example the output lines are driven 0, 1, 2 and 3 clock cycles after the *outputReady*-signal is activated. The value of *buf_00* is posted to the external bus (variable *out*) shortly after the output ready signal is activated.

Input channels must not be accessed relative to the output ready signal, since this does not make sense.

6 Related work

A widespread approach for C based silicon design is SystemC [23]. SystemC provides an interoperable modelling platform which enables the development and exchange of very fast system-level C++ models. It also provides a stable platform for development of system-level tools. SystemC provides language constructs to implement algorithms at different levels of abstraction. System engineers usually start at a very high level of abstraction. SystemC defines an abstraction level on top of the synthesisable register transfer level (RTL)[21]. The so-called Transaction-Level-Modelling[22] may be subdivided into a non-cycle-accurate (packet level) and a cycle-accurate level (cycle level).

6.1 TLM

The TLM is a very good approach for simulation and to find system bottlenecks at a very early stage of development. But, the TLM is not a synthesisable model. For real synthesisable code the TLM model has to be converted into a RTL model. The RTL model can be implemented using any HDL[23]. But, it may also be implemented in SystemC. Usually the TLM model is implemented by as System engineer whereas the RTL model is implemented by a hardware engineer with detailed knowledge of the underlying hardware. The RTL model developer has to implement the same functionality as the TLM model developer. This phase of development is supported by the SystemC RTL synthesis guideline [24].

6.1.1 A smart TLM methodology

A smart methodology which enables the designer to reason about the architecture of a very complex system was proposed by Rainer Leupers et al. [25]. It is based on the SystemC 2.0 library. The methodology also provides capabilities for co-simulating multiple levels of abstraction. The main idea of this methodology is a channel library which has been developed to enable the exploration and co-verification at two levels of abstraction:

- *Packet* level
- *Cycle* level

[21] In RTL design, a circuit's behaviour is defined in terms of the flow of signals or transfer of data between registers, and the logical operations performed on those signals. RTL is used in hardware description languages like Verilog and VHDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can then be derived.

[22] TLM emphasizes verification centric design whereby a system is composed of models that form a core reference framework within which to explore system architecture and verify functionality. A consistent TLM reference framework improves collaboration between the traditionally diverse disciplines of SW / HW design, modeling, verification and implementation. This maximizes reuse and minimizes error, leading to better designs with reduced risk.

[23] HDL = Hardware Description Language

The objects of this library are token producing and consuming processes connected to each other by intelligent channels. The processes model the functional behaviour whereas the channels take care of the specific timing. Due to this separation it is possible to map the functional correct system model easily to different target architectures.

6.1.1.1 Disadvantages As already mentioned in the last paragraph, it is still necessary to convert the high level architectural model into a lower-level RTL model in order to generate a synthesised silicon net list. This process is error-prone. It may be a big step to close this gap by an automatic tool. That means, it may be better to generate silicon net lists directly from the high-level behavioural description. This is what our work is focused on. Another approach to close this gap was proposed by Ian Page et al. [1].

6.1.1.2 Intermediate code comparison In this thesis an intermediate code is proposed that bears a resemblance to the three address code [26]. It's format consists of a sequence of statements each of which references at most three statements. It provides constructs for concurrent execution and timing. In this document we have proposed an intermediate code that can be translated efficiently to silicon net lists and to assembly language. Aother intermediate code called IR-C was presented by Rainer Leupers et al. [27]. It retains the executability[24]. The key idea is to represent the intermediate code itself in C syntax. Since the intermediate code still is a C program it can be translated to assembly language using a common C compiler.

In fact it may be possible to use IR-C as an input to our compiler, if our restrictions and extensions apply to this code. This procedure may have some advantages. First of all, the user can choose between two levels of abstraction: the top level similar to a normal programming language and the IR-C which has a lower level of abstraction (comparable to the intermediate code presented in this thesis). In both cases the same compiler may be applied. A second advantage belongs to the timing control. According to the assumption 4.2.4 all the ATCs take the same execution time. Since the modified[25] IR-C can be translated to our intermediate code representation, it is possible to control the timing at source code level. This may be helpful for problems that require high cycle accuracy. But of course, it is also possible to mix the levels of abstraction. For instance it is possible to code the IOs (which have usually high timing requirements) using IR-C and the algorithmic parts at a higher level of abstraction. Refer to the following example:
Example 59:

```
INPUT UINT8 A;
```

[24]Executability means the source code can be compiled into a machine program and executed for validation or simulation purposes.

[25]with our restrictions and extensions

```
UINT8 main( UINT8 a, UINT8 b, UINT8 c )
{
   UINT8 a, b, c, B;

   x = a+b;
   y = b*c;
   z = c-a;
   B = A+a+b+c;

   return B;
}
```

A corresponding implementation in IR-C reads as follows (refer to the left column in the following example).

Example 60:

```
IR-C                                          Our intermediate code
====                                          =====================

INPUT UINT8 A;

void main(UINT8 a_3,UINT8 b_4,UINT8 c_5)  main
{                                             (0)  a_3 <--|  b_4 <--|  c_5 <--|
   int x_7,y_8,z_9,t1,t2,t3,t4 ,t5,t7,B;

   t1 = a_3 + b_4;                            (1)  (+) t1 <-- [a_3 b_4]
   x_7 = t1 ;                                 (2)  (=) x_7 <-- [t1]
   t2 = b_4 * c_5;                            (3)  (*) t2 <-- [b_4 c_5]
   y_8 = t2 ;                                 (4)  (=) y_8 <-- [t2]
   t3 = a_3 - c_5;                            (5)  (-) t3 <-- [a_3 c_5]
   z_9 = t3 ;                                 (6)  (=) z_9 <-- [t3]
   t4 = x_7 + y_8;                            (7)  (+) t4 <-- [x_7 y_8]
   t5 = t4 + z_9;                             (8)  (+) t5 <-- [t4 z_9]
   B = A + t5;                                (9)  (+) B <-- [A t5]

   return B;                                  (10) (=) RTS_VAR_NAME <-- [B]
}                                             (11) (RTS) [RTS_VAR_NAME]
```

Our corresponding intermediate code is given in the right column of the code snippet above. One can see that it is possible to convert the IR-C code one by one to our intermediate code. Therefore each of the statements in the IR-C source code takes exactly one clock cycle.

6.2 A mature C based design system

We compare our algorithm to one proposed by Ian Page et. al. which has been implemented successfully by Celoxica Inc[26]. The software called SK1 is available on the market as a commercial product. The comparison seems to be difficult, since the two systems (ours and the Celoxica system) do not have exactly the same goal. The algorithm proposed by Ian Page is intended to enable software programmers to generate silicon devices whereas our system is intended to demonstrate an algorithm for Dual Compilation (hardware and software). Even if the two systems have this difference it is meaningful to compare them for the following reasons:

- Both systems use a sequential programming language.
- The main difficulty is the hardware compilation, since the generation of microprocessor code from a sequential description like C is a well known process.
- Even if the Celoxica suite does not generate code for microprocessors, it is possible to do this by using a different tool (normal commercial Compilers like Ms Dev Studio or Borland C).
- Both systems are producing Edif net lists. The Celoxica suite may also be used as a C to VHDL - Translator. This feature will not be discussed within this thesis, since we do not think that this is a key feature.

The following sections contain a discussion on the advantages, disadvantages, similarities and differences of the two systems. We choose the algorithm by Ian Page respectively the Celoxica design suite, since it is available on the market as a commercial product. Therefore we assume that this system implements a state-of-the-art methodology.

6.2.1 Similarities

As already mentioned in the introduction, the programming language is very similar. In particular the syntax of the design entry is almost the same. Both systems use a language derived from ANSI C, since it is wide spread in the software developers community. We assume that Ian Page and his colleagues had the same motivation to use C (they call it 'Handel C' [3]). Handel C has an additional key word which is one of the most important characteristics of silicon devices. We decided to use the same keyword for this purpose. The second common item is the compiler output. Both systems generate Edif files at net list level. The net lists are the lowest possible design level which are device-independent. It is still necessary to map the cells to a particular device and route the connections between the cells. These two final steps are highly device-dependent. On the other hand they are well known by the device vendors like Xilinx or Altera and Software for these steps is provided by them. Therefore files containing net lists are a good choice for the compiler output.

[26] Celoxica is one of the technology leader in C design and synthesis

6.2.2 Differences

The two presented systems have several differences. Therefore we have to limit the discussion to the key differences. First of all, the Celoxica design suite DK is a commercial product whereas our compiler is just a technology demonstrator. Therefore the stage of maturity is not comparable. Anyway we will focus the discussion on the differences of the underlying programming language and compiler algorithm. A compiler discussion is done briefly at the end of this chapter. We split our discussion into 5 main blocks.

- *Concept*
- *Design* entry
- *Compilation* process
- *Compiler* output
- *Implementation*

6.2.2.1 Concept The goal of Ian Pages algorithm is to generate silicon from C respectively a derivative of C. Anyway it is possible to compile Handel C for a microprocessor by using a 'normal' compiler. Therefore this method is also useable for hardware/software co-design. In contrast we developed a method which is able to compile the source code for hardware and software using the same compiler. Refer to the sketch in Fig. 27 for details.

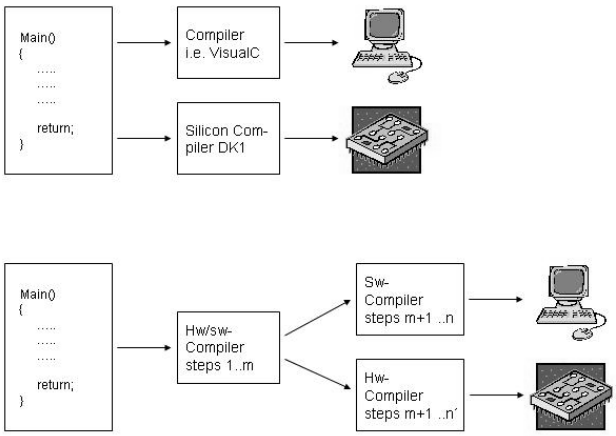


Figure 27: Conceptual difference

The two sketches above show a typical compilation process. Normally the developer will start with the software solution. When using the method by Ian Page (hence we call it the reference method) the developer is using a standard C compiler like Ms Dev Studio or Borland C. On

a computer he is able to test the programmed algorithm. If high execution speed or timing accuracy is required, he will switch to the second branch and will generate net lists that can be programmed to a programmable gate array or used to generate an ASIC - device.

In our implementation always the same tool is used for both, hardware and software. A common intermediate code is generated at the compilation step m (see Fig. 27). This code may be used for timing prediction for both compilation branches. Since this intermediate code has a very low abstraction level (it may be compared to the assembler code of a formal machine) it is possible to predict the timing behaviour even if a software target is chosen. Refer to section 4 for details on the timing of clocked circuits.

The timing itself is one of the major differences of the two concepts. The user of the Handel C compiler has to calculate the timing at source code level by himself. This procedure is quite simple. In this particular implementation, all the source code statements consume the same time. In some cases this procedure wastes time, since the physical implementation of different operations may take different operation times. Also the programmer can write statements regardless of the source code layout. See the example 59.

Example 61:

```
//.......
   int i, sum=0;

   par
   {
      for (i=0; i<10; i++)
      {
         sum += i;
      }
      x = 7;
   }
//.......
```

The two statement blocks within the par block have the same running time. One can see that the real physical implementation of the second statement ($x = 7;$) is much faster than the first one (the loop). Therefore the result of the second statement has to be latched until the result of the first one is ready for the next clock cycle. Refer to document [5] for details. In contrast to this method our timing calculator uses the intermediate code. Each of the operations in this code contains just one operation. Therefore the running time is not dependent on the source code layout. Refer to the following example.

Example 62:

```
//.......
   {
      int var0, var1, var2, var3, var4;

      sum = var0 + var1 + var2 + var3 + var4;
   }
//.......
   {
      int var0, var1, var2, var3, var4;

      sum = var0;
      sum += var1;
      sum += var2;
      sum += var3;
      sum += var4;
   }
//.......
```

We assume that the two blocks above do not run in parallel, but sequentially. One can see that the two blocks are doing the same calculation. Using our method the two blocks have the same running time. Using the reference method the running time is different. Since all the assignments take exactly one clock cycle the first one takes one clock cycle whereas the second block takes five.

A third major conceptual difference of the two design systems is the way how the synchronization of concurrent blocks is done. The reference algorithm uses synchronization signals whereas our system uses a global clock. Both methods have advantages and disadvantages. For a discussion of this item refer to chapter 6.3.

6.2.2.2 Design entry The design entry of both systems is very similar. Both systems use an algorithmic description. This is a higher level of abstraction than the register transfer or structural level which is normally used for hardware design. But, the conceptual differences mentioned in the last chapter yield some syntactic and semantics differences. The reference design uses signals between concurrent design blocks for synchronization. The programming language must provide the constructs for the signalling. These constructs are implemented similar to semaphores in common software development systems. Refer to [3] for details. Our system uses a global clock for synchronization. In contrast to the synchronization implemented with signals (semaphores) it is not possible to program a loop without a fix number of repetitions when a global clock is used.

One of the most important requirements when using a global clock is a deterministic timing behaviour. This means that the timing has to be calculable during the compilation process,

and it must not be dependent on the particular input data and/or operation conditions except the situation that these conditions are known at the compilation time. Refer to Fig. 28.

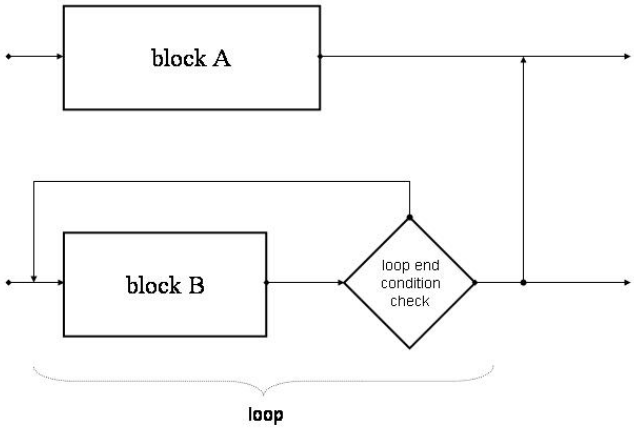


Figure 28: Loop generated by the reference design

The output data of block A a has to be latched until the signal from the second (concurrent) data pipe is sent. This signal is sent after the last iteration of block B. This is a well known method. It is used in multitasking operating systems. But this may be a non-deterministic process, since the number of iterations can be variable. Even an endless loop is programmable. In this case, the result of block A has to be latched forever. One can see that it is not possible to calculate the timing during the compilation process. It has to be measured with a particular set of input vectors and operation conditions. In contrary to this method we want to use a global clock. Using a global clock requires the timing to be calculated at the compilation time. Therefore we must not use repetitions with a variable number of iterations. In this case the user does not take care of the synchronization. Therefore synchronization signals are not required and not provided by our programming language. Refer to example 59. In this example the numbers from 0 to 9 are added in concurrence to the assignment *x=7;*. Since the values of x and the final value of *sum* are used outside the parallel block, it is necessary to latch the value of x for 9 cycles (assumed that the assignment and the addition take the same time). This is done automatically, since the process delay of each block is calculated during the compilation process. The compiler will produce one of the designs. printed in the Fig. 29 and Fig. 30. Whether the first or the second design is produced depends on the required optimization method. The first one (Fig. 29) consumes a smaller number of silicon gates whereas the second one (Fig. 30) has a higher operation speed in terms of the number of processed data sets per time unit.
The first circuit block in Fig. 29 called *D10* is a delay block. It activates the output line ten clock cycles after its input line has been activated. This is exactly the execution time of the adder (notice that the adder runs within a loop with ten iterations).
The first row in Fig. 30 is the time steering. It consists of 10 delay elements (D-flip-flops). Each of these blocks delays the processing exactly for one cycle. The second row consists of

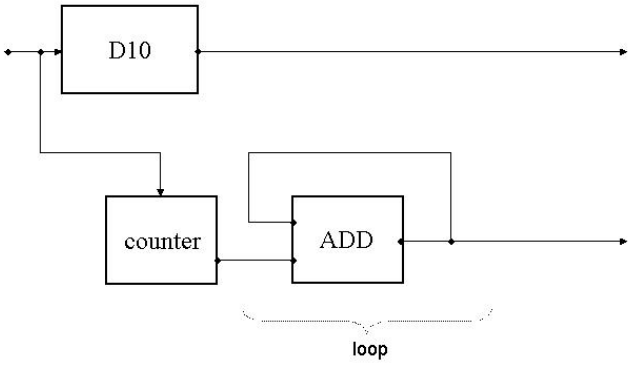


Figure 29: Loop optimized for size

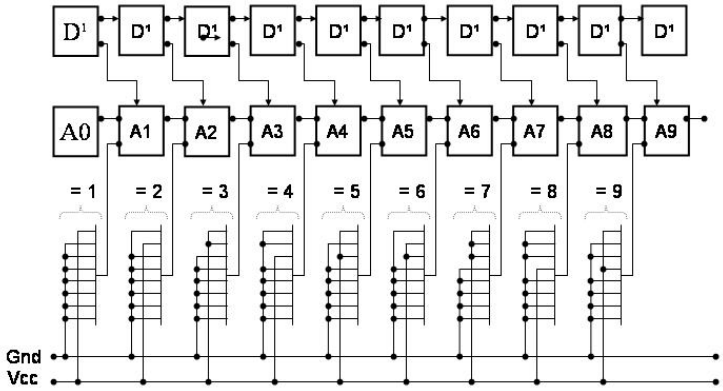


Figure 30: Loop optimized for speed

10 digital adders. Each of them takes the result of the previous block and adds the desired constant. One can see that there are no synchronization signals in the design and the delay time can be calculated during the compilation process. It is exact 10 cycles. If the reference system is used, the user will implement the following example using synchronization signals.

Example 63:

```
{
   signal sync1

   sync1 = 1;
   int i, sum=0;

   for (i=0; i<10; i++)
   {
      sum += i;
```

```
      if (x==9)
         sync1 = 0;
   }
   wait sync1;
   x = 7;
}
```

The reference code above has the same function as our code in example 59. The only difference is the synchronization signal. It is necessary, since the compiler does not calculate the timing. Our compiler inserts an adequate delay to the global clock for synchronization.

The reference compiler will produce a silicon design comparable to the sketch in Fig. 31. The circuit is not optimized for speed.

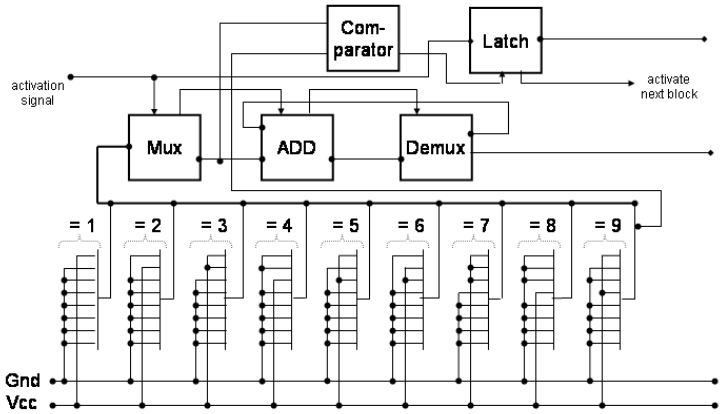


Figure 31: Reference design

The semaphore signal (on top of the schematic) is latched and the succeeding circuit blocks are blocked until the comparator unblocks them. This happens after the last iteration. Refer to document [3] for details on the Handel C design entry.

6.2.2.3 Compiler The comparison of the two compiler implementations is not one of the major goals of this thesis. As already mentioned, the DK1 suite by Celoxica Inc is a commercial product and available on the software market whereas our compiler was implemented just for demonstration purposes. Therefore we limit the discussion to the basic concepts. The DK1 suite (that incorporates the compiler) produces two different kinds of outputs. On the one hand it is able to output VHDL code at RTL level (RTL = Register Transfer Level). This can be compiled by standard design tools like ISE by Xilinx Inc (refer to document [6] for details on ISE). If this output is required, the compiler can be considered as a translator, since real silicon is not synthesized. On the other hand the DK1 suite can synthesize silicon net lists. These are stored in EDIF files.

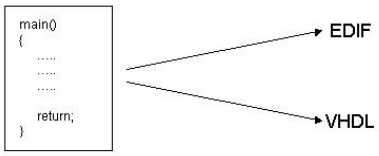


Figure 32: Reference compiler design chain

Our compiler was designed for hardware/software co-design. We do not generate VHDL code or similar hardware descriptions. RTL-VHDL is a high level description language and normally used as design entry. On the other hand, our compiler can generate microprocessor code for the ARM architecture (refer to [7] for details on the ARM architecture).

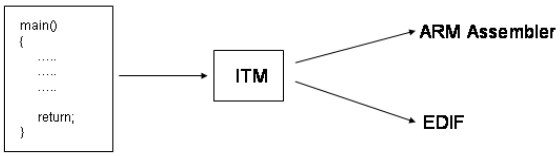


Figure 33: Our compiler design chain

The next interesting difference belongs to the timing behaviour. Our compiler incorporates a timing calculator. The reference compiler cannot calculate the timing exactly for conceptual reasons.

6.3 Advantages/Disadvantages

6.3.0.4 Design entry This is the easiest part, since there are just a few minor differences. The only mentionable once are the design elements that belong to the synchronization of the concurrent blocks and the time steering. The elements used by the reference compiler are signals (comparable to the semaphore mechanism of a common programming language). Since this method is taken from common software development systems it is probably easy to use for software programmers. Our system is based on exact timing calculation. Therefore we do not use synchronization signals. Instead we need constructs for the global clock steering. Refer to 2.2.4 for details. This procedure is used in some high level hardware description languages like VHDL (refer to [16] for details on VHDL). Software programmers are not used to these design elements. ASIC and FPGA designers are used to this procedure. Summing up, we can say that the design entry of the reference design is more software-like whereas ours is more hardware-like.

6.3.0.5 Compilation Process When using the reference method, different compilers have to be used for hardware and software target. This may cause some inconsistencies, since the design entry for hardware targets is Handel-C whereas the common software compiler inputs ANSI-C source codes. When using our algorithm, this problem is ruled out, because the same compiler is used for both targets. Refer to Fig. 34 and Fig. 35.

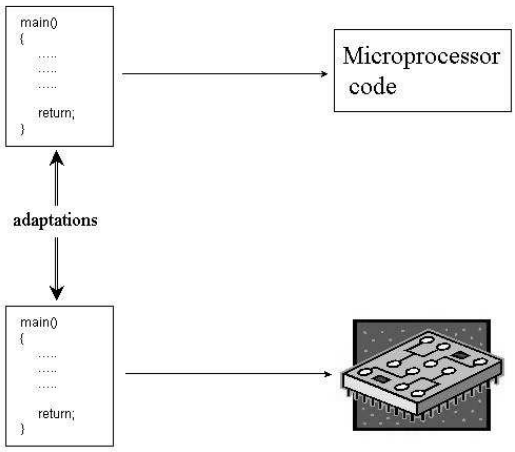


Figure 34: Reference design chain

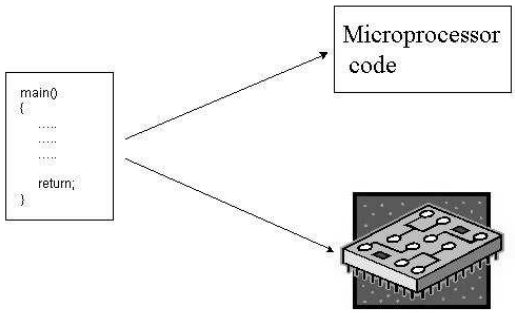


Figure 35: Our design chain

In general the developer starts with a software target (microcontroller) because these targets have better debugging and testing opportunities. If necessary, he will switch to a hardware target. If the reference method is used, it will probably be necessary to make some adaptations, since a different compiler is used. Our compiler has the same design entry for both targets. The branch is done as a later stage (see Fig. 35). This is an enormous advantage, since no adaptations are necessary when the target is switched (from a microcontroller to a silicon device). For this purpose, we have developed an intermediate code (see section 3.2.1). This

code is common for both targets. It contains atomic design blocks. They may be compared to a common assembler language. Some additional constructs are necessary for the time steering and for concurrent execution. This intermediate code is highly suitable for timing control, since the exact running time of these design blocks is known. These blocks are atomic. They have no separable substructure. See the following two simple statements.

Example 64:

```
x = a+b;
y = a*b + c*d;
```

The first statement takes 2 cycles whereas the second one takes 9 cycles[27]. The reference implementation takes the same time for both source code statements. Refer to Fig. 36.

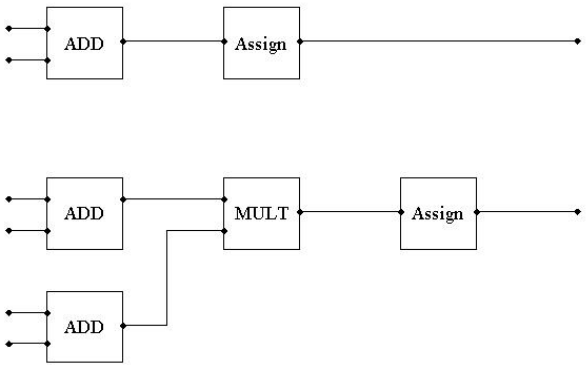


Figure 36: Running time

Of course, the calculations in the second statement take more time. Therefore the result of the calculation in the first statement has to be latched until the second one is finished.

[27]The 8 bit multiplication consumes 7 clock cycles

7 Outlook and Future Research

The main task of this thesis was to develop a programming language that is suitable for hardware/software-codesign. Since the work has a theoretical character, it was not a major aim to write a compiler that implements all the aspects of the proposed language or the silicon synthesizer and the software code generator respectively. The software written during the thesis is only for demonstration purposes. The most important work in the future may be the implementation of a smart compiler in order to evaluate the language and compiler algorithm for real hardware and software implementations.

7.1 Methodology items

One of the most important areas covered in this thesis is the timing of the synthesized silicon or the generated code respectively. The algorithm is intended to produce time-accurate designs. Especially silicon devices require this accuracy, since they are mostly used to implement real time systems (i.e. lower layers of communication devices, devices for steering and controlling etc). Therefore it is highly interesting to test the timing in real implementations. For these tests it is necessary to implement a fully functional compiler and test bench.

Another interesting point for future research is the question: Is it possible to bring the two approaches - one proposed by Ian Page et. al. and ours - together, in order to take advantage of the powers of both systems? As already mentioned in section 6.3 both approaches have advantages. The reference method for example is easier to understand for pure software developers, whereas ours implements some design elements taken from hardware description languages. If it is possible to bring together the two approaches, it will really close the gap between hardware and software development.

Our algorithm uses a mixture of automatic and manual parallelization. In order to accelerate the development it is desired that the automatic part be as large as possible. Especially when using complex data structures or non-static dereferences (refer to example below) it is very difficult to automate the parallelization. This is a very important item for future research.

Example 65:

```
input int a, b;
//............
int var[5];
{
   var[a] += a;
   var[b] += b;

   //........
```

```
}
//............
```

The example above shows a situation where the two statements may not be parallelized easily. It may be possible that *var[a]* and *var[b]* point to the same memory (if $a=b$).

7.2 Additional compiler issues

Although the compiler was developed to demonstrate the capabilities of the proposed design language, it may be extended to a fully featured development tool. In this section we present the most important things to do in order to achieve this.

7.2.1 Optimizer

How to develop a high efficiency optimizer for the software target is a well known task. Therefore we focus our interest on the hardware target.

In the current implementation the silicon optimization is done separately for each function. In many cases this does not yield an optimal circuit. Refer to the following short example:

Example 66:

```
#include "mytypes.h"

clock { frequency = 300; }

static UINT8 dummy( UINT8 A00, UINT8 A01, UINT8 A10, UINT8 A11 )
{
   UINT8 buf, buf1;

   buf = (A00*A11) - (A10*A01);
   buf1 = A00 + A01 + A10 + A11;

   return buf + buf1;
}


UINT8 main( UINT8 inp_00, UINT8 inp_01, UINT8 inp_10, UINT8 inp_11 )
{
   UINT8 buf_00;
   UINT8 buf_01;
   UINT8 buf_10;
   UINT8 buf_11;
```

```
    UINT8 det0, det1;

    /* Get the transpose of the input matrix. */
    buf_00 = inp_00;
    buf_01 = inp_10;
    buf_10 = inp_01;
    buf_11 = inp_11;

    /* Calculate the determinant of the transposed matrix. */
    det0 = dummy(inp_00, inp_01, inp_10, inp_11);
    det1 = dummy(buf_00, buf_01, buf_10, buf_11);

    return det0 + det1;
}
```

The compiler generates and optimizes the *main* function and the function *dummy* separately. The results are given in Fig. 37 and Fig. 38.

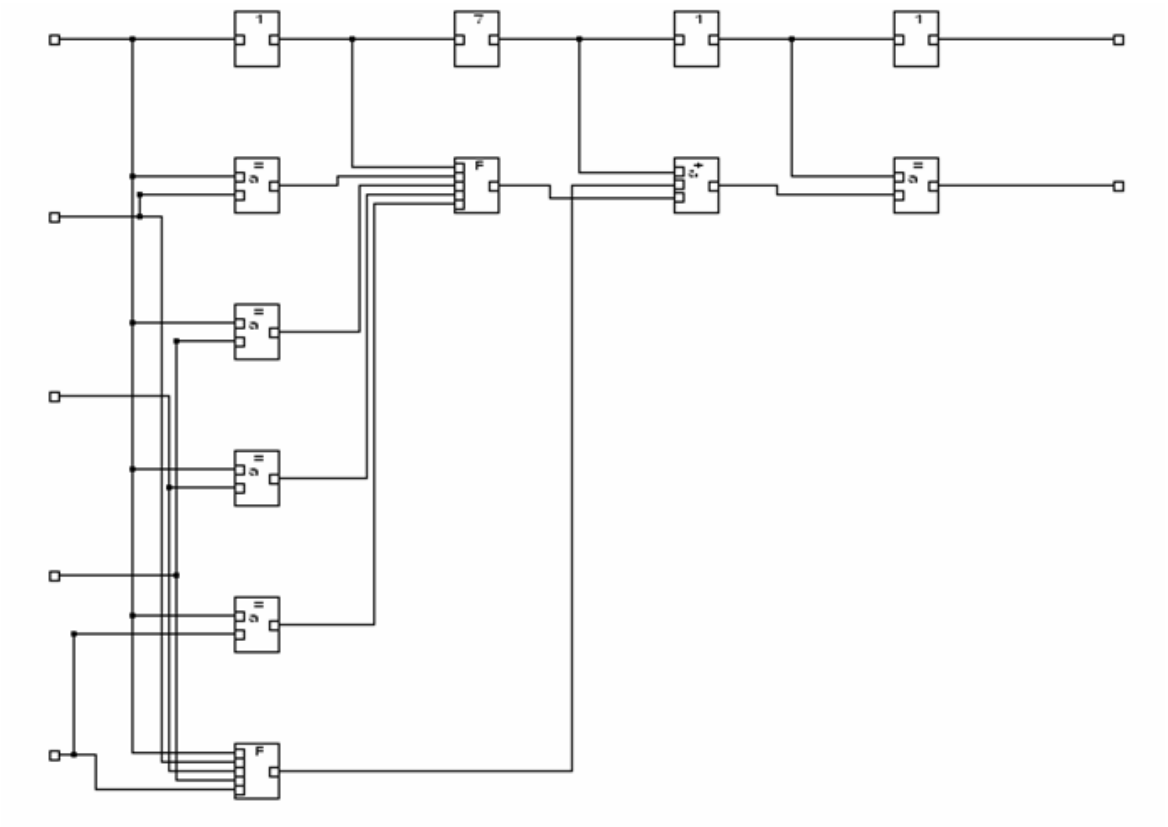


Figure 37: Separately optimized functions (main)

One can see easily that the circuit is not optimal. The addition of the four input components done within the *dummy* function may start at the beginning of the circuit. But in our imple-

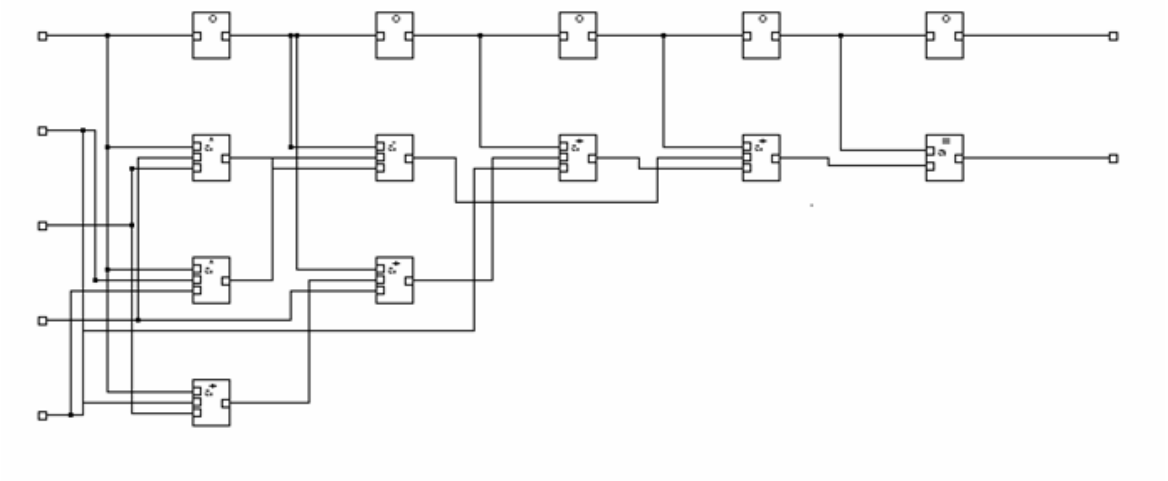


Figure 38: Separately optimized functions (func1)

mentation it starts one cycle later. The optimal circuit is given in Fig. 39.

An easy way to achieve the optimal circuit is to replace the function call by its body (refer to example 65).

Example 67:

```
UINT8 main( UINT8 inp_00, UINT8 inp_01, UINT8 inp_10, UINT8 inp_11 )
{
   UINT8 buf_00;
   UINT8 buf_01;
   UINT8 buf_10;
   UINT8 buf_11;
   UINT8 det0, det1;
   UINT8 buf, buf1;

   /* Get the transpose of the input matrix. */
   buf_00 = inp_00;
   buf_01 = inp_10;
   buf_10 = inp_01;
   buf_11 = inp_11;

   buf = (inp_00*inp_11) - (inp_10*inp_01);
   buf1 = inp_00 + inp_01 + inp_10 + inp_11;
   det0 = buf + buf1;
```

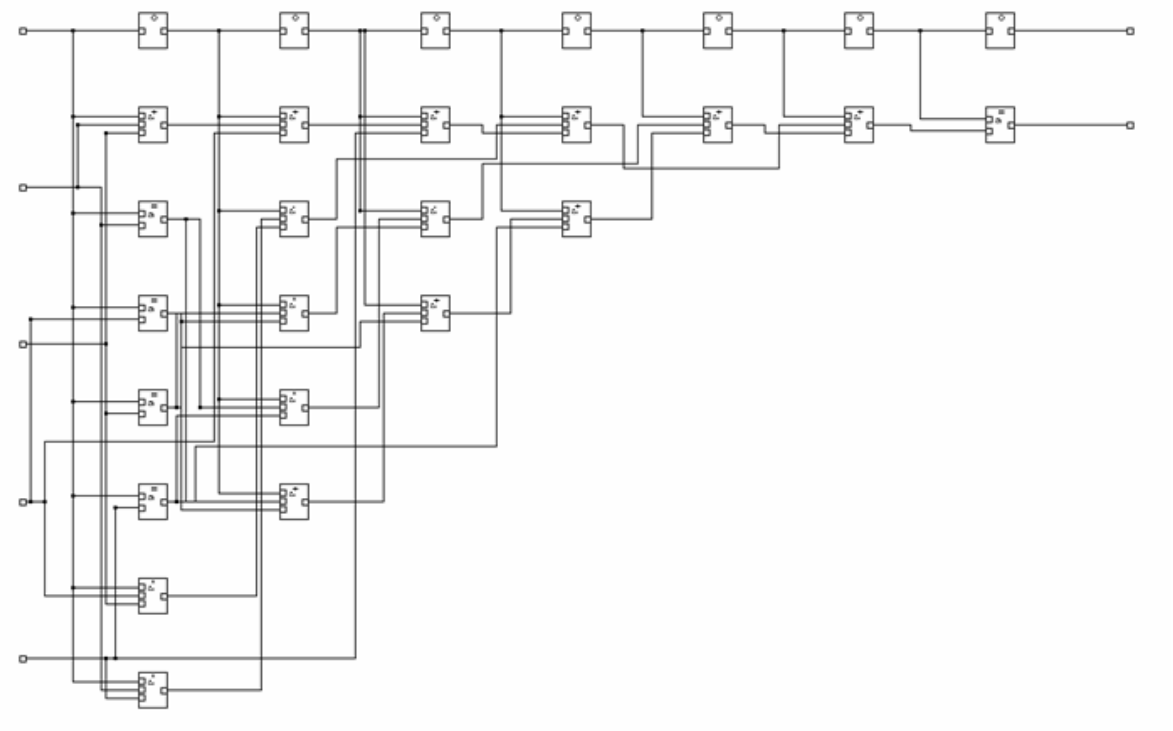


Figure 39: Optimal circuit

```
    buf = (buf_00*buf_11) - (buf_10*buf_01);
    buf1 = buf_00 + buf_01 + buf_10 + buf_11;
    det1 = buf + buf1;

    return det0 + det1;
}
```

This may generate on optimal circuit but of course, is not the best way for many reasons. First of all, function calls make the source code readable. Usually the functions are used more than once. Therefore it may be a good habit to encapsulate the code in a function.

We propose two different ways to solve this problem. The first one is taken from normal compilers. The code is usually optimized at assembler level. For example, if the function body is short and not used often it will be used as inline code. We can use a similar technique at the intermediate code level. In this case the intermediate code does not contain any subroutine calls and the design will be optimal. The disadvantage of this method is a highly increasing compilation time, since the entire optimization is done in one step. A second and probably better method is a post-link optimization. That means that a second optimization is done after the different modules are linked together.

7.2.2 Synthesizer

In the current compiler version, re-usage of circuits has not been implemented. In the example in appendix 9.2 the function that performs the matrix multiplication is invoked four times, once for each of the four components. The circuit synthesizer generates always four instances of this module that run concurrently. This method of course, has the highest performance. On the other hand it consumes a maximum silicon space. In some cases the timing requirements are weaker, or a higher clock speed is possible. In this case it may be useful to generate only one instance of the module and use it four times sequentially. This method saves silicon space. From a technical point of view this method is quite easy. The simple input wiring must be replaced by a data multiplexer and the controlling is a simple flip-flop loop. Refer to Fig. 40.

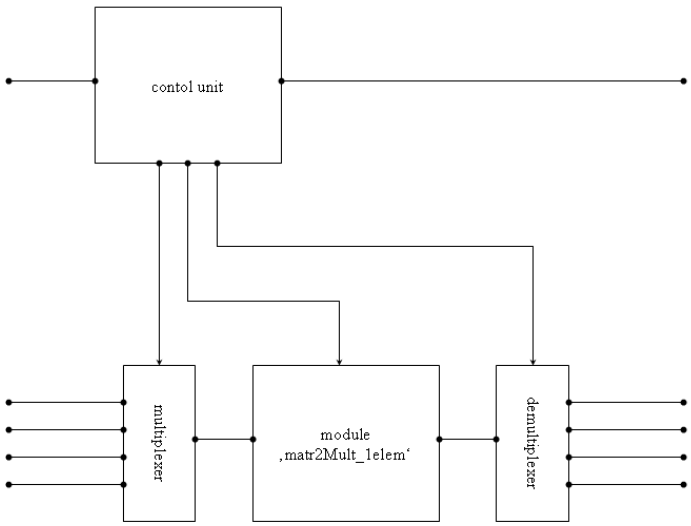


Figure 40: Module loop

The logic timing calculation is also simple. The running time is four times the running time of the module *matr2Mult_1elem*. The physical delay of the block may increase (due to an additional multiplexer and demultiplexer). Therefore the maximum clock is probably lower. But in most cases the delay of a multiplexer and a demultiplexer is very low compared to other circuits (i.e. a multiplier). Therefore we renounced to implement this feature. It does not restrict the statements of our analysis.

7.2.3 Data exchange

The input and output channels are declared as *input* and *output* variables. In real devices these channels are connected to external pins or to internal interface blocks. These blocks are usually complex circuits (i.e. buffers, input/output busses). The current version of our compiler does not provide support for implementation, debug or timing analysis of these circuit blocks. Therefore the generated design will not run on real devices without additional effort for the implementation of 'real' IOs. It will run only in simulation mode. From a design point of view this is not a restriction, since the device vendors (i.e. Altera, Xilinx) provide tools for automatic

implementation of several types of IOs. But these tools do not provide an elegant method to calculate the timing behaviour. The IO timing has to be estimated in a simulation. This, of course, is not our timing approach.

8 Summary

We have seen how a high level programming language like C may be used for hardware/software-codesign. Since the compilation for a software target is a well-known process, we have focused our discussion on the compilation for a hardware target. We started our work with the programming language proposed by Ian Page et. al. [2]. The first step was to figure out the deficiencies of this programming language (we called it the reference system in our discussion). We figured out that using the reference system prevents the developer from implementing time-accurate hardware designs. The reference design does not provide any constructs for explicit timing implementation. It assumes that all the statements at source code level use the same processing time. For this purpose we implemented a steering mechanism for the timing. Many expert groups tried to generate silicon net lists from ANSI-C but these approaches lead to inefficient silicon. We tried to find an optimal compromise between nearness to ANSI-C and the requirements for efficient silicon compilation.

The second main innovation step was the implementation of an intermediate code. We showed that this intermediate code enables the system to produce target designs with higher quality than the direct generation from the source code. An efficient intermediate code is essential for our system because we want to generate time-accurate systems. The intermediate code proposed in section 3.2.1 meets all the requirements for a precise and efficient timing. Its level of abstraction is high enough to hide the device specifics[28] and low enough to avoid any additional processing except the routing and placing that is done by the software provided by the device vendors. The intermediate code is also suitable for high-performance optimizations, especially for timing optimizations.

Common hardware and software design systems do not take care of the timing analysis during compilation time. The developers of these systems assume that the user measures the timing after the design is finished. This is done using simulators at early stages of the design. In later stages it may be done on the real target. If the design does not meet the requirements it will be refined. This design loop has to be repeated until the results meet the desired parameters. We tried to go a different way and calculate the timing during compilation time. This, of course, is done at intermediate code level. Even if the intermediate code is device-independent, the result of the timing calculation is not. It depends on the particular hardware or software device that is used in the final target. For example, if we use an FPGA as a particular target the timing depends on one hand on the result of the placing and routing and on the other hand on the latencies of the particular implementation of the Boolean functions. The timing information is provided by the device vendor (i.e. the latencies of the different atomic design blocks.

The compiler proposed in this thesis is not intended to be finished within the scope of this thesis. It was implemented just for demonstration purposes. The main aim of this program is

[28] That means the exact implementation at the gate level.

to demonstrate that it is possible to implement a compiler for the proposed algorithm.

A comparison to a similar work has shown that our approach has a lot of advantages. The main ones are

- Our algorithm can produce hardware and software from the same source code.
- Our algorithm can produce time-accurate designs.
- The designer is not burdened to take care of the synchronization of the different data paths.
- The optimization and refinement is done at a suitable stage - the intermediate code (a final optimization is also done at the assembler level or the net list level respectively).
- It is not necessary to measure the timing after finishing the design, since it is calculated in the compilation process.

9 Appendix

9.1 Proof of Equation 4 by complete induction

$$\text{Assume:} \quad F(ATC_1^1) = F(ATC_2^1) \tag{14}$$

$$\Rightarrow A * \sum_{n=1}^{1} t_{i_1} + \underbrace{B * \sum_{n=1}^{1} \max_{n \neq m} (t_{i_1}^n - t_i^m)^2}_{=0} =$$

$$A * \sum_{n=1}^{1} t_{i_2} + \underbrace{B * \sum_{n=1}^{1} \max_{n \neq m} (t_{i_2}^n - t_i^m)^2}_{=0} \tag{15}$$

$$\Rightarrow t_{i_1}^1 = t_{i_2}^1 \quad \Rightarrow \underline{\underline{ATC_1^1 = ATC_2^1}} \tag{16}$$

The equation above says that the two configurations ATC_1^1 and ATC_2^1 are identical.

$$\text{Assume:} \quad ATC_1^N = ATC_2^N \tag{17}$$

$$\begin{aligned} &\text{to show:} \quad ATC_1^{N+1} = ATC_2^{N+1} \\ &\text{if} \quad F(ATC_1^{N+1}) = F(ATC_2^{N+1}) \end{aligned} \tag{18}$$

$$\begin{aligned} A * \sum_{n=1}^{N+1} t_{i_1} + B * \sum_{n=1}^{N+1} \max_{n \neq m} (t_{i_1}^n - t_i^m)^2 = \\ A * \sum_{n=1}^{N+1} t_{i_2} + B * \sum_{n=1}^{N+1} \max_{n \neq m} (t_{i_2}^n - t_i^m)^2 \end{aligned} \tag{19}$$

$$\begin{aligned} A * \sum_{n=1}^{N+1} t_{i_1} + B * \sum_{n=1}^{N+1} \max_{n \neq m} (t_{i_1}^n - t_i^m)^2 - \\ A * \sum_{n=1}^{N+1} t_{i_2} + B * \sum_{n=1}^{N+1} \max_{n \neq m} (t_{i_2}^n - t_i^m)^2 = 0 \end{aligned} \tag{20}$$

$$\begin{aligned} &\text{From (17) it follows} \quad A * \left(t_{i_1}^{N+1} - t_{i_2}^{N+1} \right) + \\ &B * \left(\max_{m \neq N+1} (t_{i_1}^{N+1} - t_i^m)^2 - \max_{m \neq N+1} (t_{i_2}^{N+1} - t_i^m)^2 \right) = 0 \end{aligned}$$

$$\text{Assume:} \quad A > 0 \quad \text{and} \quad B > 0 \quad \text{and} \quad t_{i_1}^{N+1} > t_{i_2}^{N+1}$$

$$\begin{aligned} &\Rightarrow A * \left(t_{i_1}^{N+1} - t_{i_2}^{N+1} \right) > 0 \\ &B * \left(\max_{m \neq N+1} (t_{i_1}^{N+1} - t_i^m)^2 - \max_{m \neq N+1} (t_{i_2}^{N+1} - t_i^m)^2 \right) = 0 \end{aligned}$$

$\Rightarrow$ Therefore the assumption $t_{i_1}^{N+1} > t_{i_2}^{N+1}$ is wrong

Analogously is possible to show that

$$t_{i_1}^{N+1} < t_{i_2}^{N+1} \quad \text{is wrong} \tag{21}$$

$$\Rightarrow t_{i_1}^{N+1} = t_{i_2}^{N+1} \quad \Rightarrow \underline{\underline{ATC_1^{N+1} = ATC_2^{N+1}}} \tag{22}$$

9.2 Source code of the matrix multiplication

The following code is used for most of the demo examples. It multiplies two matrices and calculates the determinant of the resulting matrix.

```
/*****************************************************************************/
/*                            LOCAL FUNCTION DECLARATION                     */
/*                                                                           */
/*  Filename    : matr2Mult.hc                                               */
/*  Version     : 1.10                                                       */
/*  Date        : 20.07.2005                                                 */
/*  Author      : Ewald Frensch                                              */
/*                                                                           */
/*  Description : Calculate the matrix multiplication of two 2x2 matrices and */
/*                the determinant of the new matrix                          */
/*                                                                           */
/*  CHANGE_NOTES                                                             */
/*                                                                           */
/*  date        name    change                                               */
/*  ------------------------------------------------------------------------ */
/*  01.08.2005  EF      Initial version                                      */
/*                                                                           */
/*****************************************************************************/

#include "mytypes.h"


/* Interface objects. Use only in main function. */
#ifdef CCHW_TRG_NL
OUTPUT UINT8 out_00, out_01, out_10, out_11;
INPUT UINT8 inp1_00, inp1_01, inp1_10, inp1_11;
#endif


static UINT8 matr2_Det
  (
    UINT8 A00, UINT8 A01,
    UINT8 A10, UINT8 A11
  )
```

```
/*----------------------------------------------------------------------------
** Function: matr2_Det
**          Calculate the determinant of a 2x2 matrix
**
** possible return values: determinant of the matrix
**
----------------------------------------------------------------------------*/
{
   UINT8 buf1;

   buf1 = (A00*A11) - (A10*A01);

   return buf1;
}



static UINT8 matr2Mult_1elem
  (
    UINT8 A0, UINT8 A1,
    UINT8 B0, UINT8 B1
  )
/*----------------------------------------------------------------------------
** Function: matr2Mult_1elem
**          Calculate one element of a matrix multiplication
**
** possible return values: Calculated element
**
----------------------------------------------------------------------------*/
{
   UINT8 buf;

   buf = (A0*B0) + (A1*B1);

   return buf;
}



#ifdef CCHW_TRG_NL
UINT8 main
  (
     UINT8 inp0_00,
     UINT8 inp0_01,
     UINT8 inp0_10,
     UINT8 inp0_11
  )
#endif
#ifndef CCHW_TRG_NL
```

```
UINT8 matr2Mult
  (
     UINT8 inp0_00,
     UINT8 inp0_01,
     UINT8 inp0_10,
     UINT8 inp0_11,
     UINT8 inp1_00,
     UINT8 inp1_01,
     UINT8 inp1_10,
     UINT8 inp1_11
  )
#endif
/*-----------------------------------------------------------------------------
** Function: main
**          Calculate the matrix multiplication of two 2x2 matrices
**          Calculate the determinant of the new matrix
**
** possible return values: determinant
**
-----------------------------------------------------------------------------*/
{
   UINT8 buf_00;
   UINT8 buf_01;
   UINT8 buf_10;
   UINT8 buf_11;
#ifndef CCHW_TRG_NL
   UINT8 out_00, out_01, out_10, out_11;
#endif

   buf_00 = matr2Mult_1elem(inp0_00, inp0_01, inp1_00, inp1_10);
   buf_01 = matr2Mult_1elem(inp0_00, inp0_01, inp1_01, inp1_11);
   buf_10 = matr2Mult_1elem(inp0_10, inp0_11, inp1_00, inp1_10);
   buf_11 = matr2Mult_1elem(inp0_10, inp0_11, inp1_01, inp1_11);

   out_00 = buf_00;
   out_01 = buf_01;
   out_10 = buf_10;
   out_11 = buf_11;

   return matr2_Det(buf_00, buf_01, buf_10, buf_11);
}
```

9.3 ARM assembler code of the matrix multiplication

```
        EXPORT matr2Mult

        CODE32
```

```
        AREA |C$$code|, CODE, READONLY

|x$codeseg| DATA

;static UINT8 matr2_Det
;  (
;    UINT8 A00, UINT8 A01,
;    UINT8 A10, UINT8 A11
;  )
matr2_Det PROC
        STMDB   SP!,{R0-R2,LR}          ;Save the used registers

;   buf1 = (A00*A11) - (A10*A01);
        MUL     R1,R2,R1
        LDR     R2,[SP,#0x0]            ;Get the parameter from stack
        MUL     R0,R2,R0
        SUB     R0,R0,R1
;   return buf1;
|LABEL.matr2_Det.4|

        LDMIA   SP!,{R0-R2,LR}          ;Restore the used registers
        BX      LR              ;Return from subroutine
        ENDP

;static UINT8 matr2Mult_1elem
;  (
;    UINT8 A0, UINT8 A1,
;    UINT8 B0, UINT8 B1
;  )
matr2Mult_1elem PROC
        STMDB   SP!,{R0-R3,LR}          ;Save the used registers

;   buf = (A0*B0) + (A1*B1);
        LDR     R3,[SP,#0x0]            ;Get the parameter from stack
        MUL     R1,R3,R1
        MUL     R0,R2,R0
        ADD     R0,R0,R1
;   return buf;
|LABEL.matr2Mult_1elem.4|

        LDMIA   SP!,{R0-R3,LR}          ;Restore the used registers
        BX      LR              ;Return from subroutine
        ENDP

;UINT8 matr2Mult
;  (
;     UINT8 inp0_00,
;     UINT8 inp0_01,
;     UINT8 inp0_10,
```

```
;     UINT8 inp0_11,
;     UINT8 inp1_00,
;     UINT8 inp1_01,
;     UINT8 inp1_10,
;     UINT8 inp1_11
;  )
matr2Mult PROC
        STMDB   SP!,{R0-R8,LR}          ;Save the used registers
        SUB     SP,SP,#0x14             ;Decrement the stack pointer

;   buf_00 = matr2Mult_1elem(inp0_00, inp0_01, inp1_00, inp1_10);
        MOV     R6,R0           ;Claim the parameter register
        MOV     R7,R1           ;Claim the parameter register
        LDR     R3,[SP,#0x4]            ;Get the parameter from stack
        MOV     R8,R2           ;Claim the parameter register
        MOV     R2,R3           ;Store the parameter to the desired parameter registers
        LDR     R4,[SP,#0xC]            ;Get the parameter from stack
        STR     R4,[SP,#0x0]            ;Store the parameter to stack
        BL      matr2Mult_1elem         ;Branch to subroutine
        MOV     R5,R0           ;Store the return value to the desired register
;   buf_01 = matr2Mult_1elem(inp0_00, inp0_01, inp1_01, inp1_11);
        MOV     R0,R6           ;Store the parameter to the desired parameter registers
        MOV     R1,R7           ;Store the parameter to the desired parameter registers
        LDR     R6,[SP,#0x8]            ;Get the parameter from stack
        LDR     R6,[SP,#0x8]            ;Get the parameter from stack
        MOV     R2,R6           ;Store the parameter to the desired parameter registers
        LDR     R7,[SP,#0x10]           ;Get the parameter from stack
        STR     R7,[SP,#0x0]            ;Store the parameter to stack
        BL      matr2Mult_1elem         ;Branch to subroutine
;   buf_10 = matr2Mult_1elem(inp0_10, inp0_11, inp1_00, inp1_10);
        MOV     R4,R0           ;Claim the parameter register
        MOV     R0,R8           ;Store the parameter to the desired parameter registers
        LDR     R1,[SP,#0x0]            ;Get the parameter from stack
        LDR     R1,[SP,#0x0]            ;Get the parameter from stack
        MOV     R2,R3           ;Store the parameter to the desired parameter registers
        STR     R4,[SP,#0x0]            ;Store the parameter to stack
        BL      matr2Mult_1elem         ;Branch to subroutine
        MOV     R3,R0           ;Store the return value to the desired register
;   buf_11 = matr2Mult_1elem(inp0_10, inp0_11, inp1_01, inp1_11);
        MOV     R0,R8           ;Store the parameter to the desired parameter registers
        MOV     R6,R1           ;Claim the parameter register
        MOV     R2,R6           ;Store the parameter to the desired parameter registers
        STR     R7,[SP,#0x0]            ;Store the parameter to stack
        BL      matr2Mult_1elem         ;Branch to subroutine
        MOV     R6,R0           ;Store the return value to the desired register
;   out_00 = buf_00;
        MOV     R2,R5
;   out_01 = buf_01;
        MOV     R2,R4
```

```
;   out_10 = buf_10;
        MOV     R2,R3
;   out_11 = buf_11;
        MOV     R2,R6
;   return matr2_Det(buf_00, buf_01, buf_10, buf_11);
        MOV     R0,R5           ;Store the parameter to the desired parameter registers
        MOV     R1,R4           ;Store the parameter to the desired parameter registers
        MOV     R2,R3           ;Store the parameter to the desired parameter registers
        STR     R6,[SP,#0x0]            ;Store the parameter to stack
        BL      matr2_Det               ;Branch to subroutine
|LABEL.matr2Mult.10|

        ADD     SP,SP,#0x14             ;Increment the stack pointer
        LDMIA   SP!,{R0-R8,LR}          ;Restore the used registers
        BX      LR              ;Return from subroutine
        ENDP

        AREA |C$$data|, DATA

|x$dataseg|

        END
```

9.4 Test bench for the model simulation of the matrix multiplication

The following test bench was used for the simulation of the hardware implementation of the matrix multiplication. It is written in VHDL. Please refer to [19] for details on how to write ModelSim test benches.

```
LIBRARY ieee  ;
LIBRARY simprim  ;
USE ieee.std_logic_1164.all  ;
USE ieee.std_logic_arith.all ;
USE simprim.vcomponents.all  ;
USE simprim.vpackage.all  ;
ENTITY matr2Mult_tb  IS
END ;

ARCHITECTURE matr2Mult_tb_arch OF matr2Mult_tb IS
  SIGNAL inp0_00 : std_logic_vector(7 downto 0);
  SIGNAL inp0_01 : std_logic_vector(7 downto 0);
  SIGNAL inp0_10 : std_logic_vector(7 downto 0);
  SIGNAL inp0_11 : std_logic_vector(7 downto 0);
  SIGNAL inp1_00 : std_logic_vector(7 downto 0);
  SIGNAL inp1_01 : std_logic_vector(7 downto 0);
  SIGNAL inp1_10 : std_logic_vector(7 downto 0);
  SIGNAL inp1_11 : std_logic_vector(7 downto 0);
  SIGNAL outp : std_logic_vector(7 downto 0);
  SIGNAL out_00 : std_logic_vector(7 downto 0);
```

```
SIGNAL out_01 : std_logic_vector(7 downto 0);
SIGNAL out_10 : std_logic_vector(7 downto 0);
SIGNAL out_11 : std_logic_vector(7 downto 0);
SIGNAL act   :  std_logic := '0'  ;
SIGNAL vcc   :  std_logic := '1'  ;
SIGNAL act_out   :  std_logic  ;
SIGNAL gnd   :  std_logic := '0'  ;
SIGNAL clk   :  std_logic := '1'  ;

COMPONENT matr2Mult
  PORT (
    gnd  : in std_logic ;
    vcc  : in std_logic ;
    clk  : in std_logic ;
    act  : in std_logic ;
    inp0_00_0  : in std_logic ;
    inp0_00_1  : in std_logic ;
    inp0_00_2  : in std_logic ;
    inp0_00_3  : in std_logic ;
    inp0_00_4  : in std_logic ;
    inp0_00_5  : in std_logic ;
    inp0_00_6  : in std_logic ;
    inp0_00_7  : in std_logic ;
    inp0_01_0  : in std_logic ;
    inp0_01_1  : in std_logic ;
    inp0_01_2  : in std_logic ;
    inp0_01_3  : in std_logic ;
    inp0_01_4  : in std_logic ;
    inp0_01_5  : in std_logic ;
    inp0_01_6  : in std_logic ;
    inp0_01_7  : in std_logic ;
    inp0_10_0  : in std_logic ;
    inp0_10_1  : in std_logic ;
    inp0_10_2  : in std_logic ;
    inp0_10_3  : in std_logic ;
    inp0_10_4  : in std_logic ;
    inp0_10_5  : in std_logic ;
    inp0_10_6  : in std_logic ;
    inp0_10_7  : in std_logic ;
    inp0_11_0  : in std_logic ;
    inp0_11_1  : in std_logic ;
    inp0_11_2  : in std_logic ;
    inp0_11_3  : in std_logic ;
    inp0_11_4  : in std_logic ;
    inp0_11_5  : in std_logic ;
    inp0_11_6  : in std_logic ;
    inp0_11_7  : in std_logic ;
    inp1_00_0  : in std_logic ;
    inp1_00_1  : in std_logic ;
```

```
inp1_00_2  : in std_logic ;
inp1_00_3  : in std_logic ;
inp1_00_4  : in std_logic ;
inp1_00_5  : in std_logic ;
inp1_00_6  : in std_logic ;
inp1_00_7  : in std_logic ;
inp1_01_0  : in std_logic ;
inp1_01_1  : in std_logic ;
inp1_01_2  : in std_logic ;
inp1_01_3  : in std_logic ;
inp1_01_4  : in std_logic ;
inp1_01_5  : in std_logic ;
inp1_01_6  : in std_logic ;
inp1_01_7  : in std_logic ;
inp1_10_0  : in std_logic ;
inp1_10_1  : in std_logic ;
inp1_10_2  : in std_logic ;
inp1_10_3  : in std_logic ;
inp1_10_4  : in std_logic ;
inp1_10_5  : in std_logic ;
inp1_10_6  : in std_logic ;
inp1_10_7  : in std_logic ;
inp1_11_0  : in std_logic ;
inp1_11_1  : in std_logic ;
inp1_11_2  : in std_logic ;
inp1_11_3  : in std_logic ;
inp1_11_4  : in std_logic ;
inp1_11_5  : in std_logic ;
inp1_11_6  : in std_logic ;
inp1_11_7  : in std_logic ;
rts_var_name_0  : out std_logic ;
rts_var_name_1  : out std_logic ;
rts_var_name_2  : out std_logic ;
rts_var_name_3  : out std_logic ;
rts_var_name_4  : out std_logic ;
rts_var_name_5  : out std_logic ;
rts_var_name_6  : out std_logic ;
rts_var_name_7  : out std_logic ;
out_00_0  : out std_logic ;
out_00_1  : out std_logic ;
out_00_2  : out std_logic ;
out_00_3  : out std_logic ;
out_00_4  : out std_logic ;
out_00_5  : out std_logic ;
out_00_6  : out std_logic ;
out_00_7  : out std_logic ;
out_01_0  : out std_logic ;
out_01_1  : out std_logic ;
out_01_2  : out std_logic ;
```

```
      out_01_3  : out std_logic ;
      out_01_4  : out std_logic ;
      out_01_5  : out std_logic ;
      out_01_6  : out std_logic ;
      out_01_7  : out std_logic ;
      out_10_0  : out std_logic ;
      out_10_1  : out std_logic ;
      out_10_2  : out std_logic ;
      out_10_3  : out std_logic ;
      out_10_4  : out std_logic ;
      out_10_5  : out std_logic ;
      out_10_6  : out std_logic ;
      out_10_7  : out std_logic ;
      out_11_0  : out std_logic ;
      out_11_1  : out std_logic ;
      out_11_2  : out std_logic ;
      out_11_3  : out std_logic ;
      out_11_4  : out std_logic ;
      out_11_5  : out std_logic ;
      out_11_6  : out std_logic ;
      out_11_7  : out std_logic ;
      act_out  : out std_logic );
  END COMPONENT ;
BEGIN
  DUT  : matr2Mult
    PORT MAP (
      gnd   => gnd,
      vcc   => vcc,
      clk   => clk,
      act   => act,
      inp0_00_0 => inp0_00(0),
      inp0_00_1 => inp0_00(1),
      inp0_00_2 => inp0_00(2),
      inp0_00_3 => inp0_00(3),
      inp0_00_4 => inp0_00(4),
      inp0_00_5 => inp0_00(5),
      inp0_00_6 => inp0_00(6),
      inp0_00_7 => inp0_00(7),
      inp0_01_0 => inp0_01(0),
      inp0_01_1 => inp0_01(1),
      inp0_01_2 => inp0_01(2),
      inp0_01_3 => inp0_01(3),
      inp0_01_4 => inp0_01(4),
      inp0_01_5 => inp0_01(5),
      inp0_01_6 => inp0_01(6),
      inp0_01_7 => inp0_01(7),
      inp0_10_0 => inp0_10(0),
      inp0_10_1 => inp0_10(1),
      inp0_10_2 => inp0_10(2),
```

```
inp0_10_3 => inp0_10(3),
inp0_10_4 => inp0_10(4),
inp0_10_5 => inp0_10(5),
inp0_10_6 => inp0_10(6),
inp0_10_7 => inp0_10(7),
inp0_11_0 => inp0_11(0),
inp0_11_1 => inp0_11(1),
inp0_11_2 => inp0_11(2),
inp0_11_3 => inp0_11(3),
inp0_11_4 => inp0_11(4),
inp0_11_5 => inp0_11(5),
inp0_11_6 => inp0_11(6),
inp0_11_7 => inp0_11(7),
inp1_00_0 => inp1_00(0),
inp1_00_1 => inp1_00(1),
inp1_00_2 => inp1_00(2),
inp1_00_3 => inp1_00(3),
inp1_00_4 => inp1_00(4),
inp1_00_5 => inp1_00(5),
inp1_00_6 => inp1_00(6),
inp1_00_7 => inp1_00(7),
inp1_01_0 => inp1_01(0),
inp1_01_1 => inp1_01(1),
inp1_01_2 => inp1_01(2),
inp1_01_3 => inp1_01(3),
inp1_01_4 => inp1_01(4),
inp1_01_5 => inp1_01(5),
inp1_01_6 => inp1_01(6),
inp1_01_7 => inp1_01(7),
inp1_10_0 => inp1_10(0),
inp1_10_1 => inp1_10(1),
inp1_10_2 => inp1_10(2),
inp1_10_3 => inp1_10(3),
inp1_10_4 => inp1_10(4),
inp1_10_5 => inp1_10(5),
inp1_10_6 => inp1_10(6),
inp1_10_7 => inp1_10(7),
inp1_11_0 => inp1_11(0),
inp1_11_1 => inp1_11(1),
inp1_11_2 => inp1_11(2),
inp1_11_3 => inp1_11(3),
inp1_11_4 => inp1_11(4),
inp1_11_5 => inp1_11(5),
inp1_11_6 => inp1_11(6),
inp1_11_7 => inp1_11(7),
rts_var_name_0 => outp(0),
rts_var_name_1 => outp(1),
rts_var_name_2 => outp(2),
rts_var_name_3 => outp(3),
```

```
      rts_var_name_4 => outp(4),
      rts_var_name_5 => outp(5),
      rts_var_name_6 => outp(6),
      rts_var_name_7 => outp(7),
      out_00_0 => out_00(0),
      out_00_1 => out_00(1),
      out_00_2 => out_00(2),
      out_00_3 => out_00(3),
      out_00_4 => out_00(4),
      out_00_5 => out_00(5),
      out_00_6 => out_00(6),
      out_00_7 => out_00(7),
      out_01_0 => out_01(0),
      out_01_1 => out_01(1),
      out_01_2 => out_01(2),
      out_01_3 => out_01(3),
      out_01_4 => out_01(4),
      out_01_5 => out_01(5),
      out_01_6 => out_01(6),
      out_01_7 => out_01(7),
      out_10_0 => out_10(0),
      out_10_1 => out_10(1),
      out_10_2 => out_10(2),
      out_10_3 => out_10(3),
      out_10_4 => out_10(4),
      out_10_5 => out_10(5),
      out_10_6 => out_10(6),
      out_10_7 => out_10(7),
      out_11_0 => out_11(0),
      out_11_1 => out_11(1),
      out_11_2 => out_11(2),
      out_11_3 => out_11(3),
      out_11_4 => out_11(4),
      out_11_5 => out_11(5),
      out_11_6 => out_11(6),
      out_11_7 => out_11(7),
      act_out => act_out  );

clk <= not clk after 5 ns;

act_force: PROCESS
BEGIN
act <= '0';
wait for 100 ns;
inp0_00 <= conv_std_logic_vector(2, 8);
inp0_01 <= conv_std_logic_vector(3, 8);
inp0_10 <= conv_std_logic_vector(4, 8);
inp0_11 <= conv_std_logic_vector(5, 8);
inp1_00 <= conv_std_logic_vector(6, 8);
```

```
inp1_01 <= conv_std_logic_vector(7, 8);
inp1_10 <= conv_std_logic_vector(8, 8);
inp1_11 <= conv_std_logic_vector(9, 8);
act <= '1';
--wait for 20 ns;
--act <= '0';
WAIT;
END PROCESS;
END ;
```

9.5 The structure of the internal net list files

The following list is the structure of the internally used net list files. They are inputs to the linker if the hardware target is selected. We use the C notation.

```
{
   typedef struct
   {
      char KeyWord[MAX_NAME_LENGTH];  // file Keyword
      int fct_Nr;  // number of functions
      T_ClockInfo globalClock;  // clocking information
      int fct_symIdx[FCT_NR];  // index of function within the symbol table
      int fct_entries_Nr[FCT_NR];  // number of entries per function
      T_atm_cmd ATCs[FCT_NR][ENTRIES_NR][ATCs_Nr];  // all the ATCs
   }
   T_propNetList;
}
```

The *KeyWord* and the *fct_Nr* have already been explained in the linker section (refer to 3.10.2). The component *globalClock* is a structure that contains all the global clocking information (i.e. the frequency, which pin is used to connect the oscillator). The *fct_symIdx* points to the name of each function within the symbol table. The symbols are stored in a different file, since they are important only for debugging purposes. *fct_entries_Nr* are the numbers of rows in the time slot matrix for each function. *T_atm_cmd* is the time slot matrix.

References

[1] Ian Page, *Closing the gap between hardware and software: Hardware-software co-synthesis at Oxford*, in 'Hardware-software co-synthesis for reconfigurable systems' number 96/036 in Colloquium: Professional Group C2 IEE, Feb 1996

[2] Ian Page, *Design Of Future Systems* p. 343, Design Automation and Test in Europe (DATE '98), 1998.

[3] Celoxica, *Handel-C Language Reference Manual, Document number: RM-1003-4.2*

[4] Celoxica, *The Technology Behind DK1* v. 1.1 August 2002

[5] Celoxica, *DK Design Suite Datasheet*, Version 1.3, 2005

[6] Xilinx, *Xilinx ISE7 Software Manuals And Help*

[7] ARM Limited, *ARM7TDMI* Technical Reference Manual

[8] ARM Limited, *ARM Instruction Set Quick Reference Card* v2.1, Ref: QRC0001H, Issued: October 2003

[9] ARM Limited, *ARM Developer Suite* - Version 1.2 - Developer Guide, Ref: DUI0056D, Issued: 23 November 2001

[10] ARM Limited, *SDT2.50 Reference Manual* Ref: DUI0041C, Issued: 18 November 1998

[11] Kernighan, Ritchie *Programmieren in C*

[12] Aho, Sethi, Ullman *Compilerbau*, 1997.

[13] John R Levine, Tony Mason, Doug Brown *UNIX programming tools.*

[14] Bronstein, Semendjajew, Musiol, Mühling *Taschenbuch der Mathematik.*

[15] Alvis J. Evans, *Basic Digital Electronics.*

[16] Maeder, VHDL *Basic Digital Electronics.*

[17] EDIF org. *EDIF Version 2 0 0*, EIA-548

[18] Xilinx *ISE*, Development System Reference Guide

[19] Mentor Graphics *Mentor Graphics Interface Guide*,

[20] Prof. Dr.-Ing. Wolfram H. Glauert *Very High Speed Integrated Circuit Hardware Description Language*, Universität Erlangen-Nürnberg, Lehrstuhl fr Rechnergestützten Schaltungsentwurf

[21] Dirk Eisenbiegler *Ein Kalkül für die Formale Schaltungssynthese*, 1999, 3-89722-197-7

[22] P. Gutberlet, H. Krämer, W. Rosenstiel *CASCH, ein Scheduling-Algorithmus für 'High-Level-Synthese*, Forschungsbereich Automatisierung des Schaltungsentwurfs (Prof. D. Schmid), Forschungszentrum Informatik an der Universität Karlsruhe

[23] SystemC Initiative *http://www.systemc.org*

[24] V1.0, Synopsys Inc., USA, *Describing Synthesizable RTL in SystemC*

[25] Tim Kogel, Andreas Wieferink, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Integrated Signal Processing Systems, Aachen University of Technology, Germany; Denis Bussaglia, Manoj Ariyamparambath Intellectual Property and Design Services, Synopsys, Inc. *Virtual Architecture Mapping: A SystemC based Methodology for Architectural Exploration of System-on-Chip Designs*

[26] Morgan Kaufmann Publishers, *1997 S.S. Muchnik: Advanced Compiler Design & Implementation*

[27] Rainer Leupers, Oliver Wahlen, Manuel Hohenauer, Tim Kogel, Aachen University of Technology (RWTH), Integrated Signal Processing Systems; Peter Marwedel, University of Dortmund, Dept. of Computer Science 12, *An Executable Intermediate Representation for Retargetable Compilation and High-Level Code Optimization*

Lebenslauf

Name:	Ewald Frensch
Geboren:	25.05.1970
Familienstand:	verheiratet mit Monika Frensch geb. Brandmüller
Kinder:	eine Tochter, Daniela Frensch, geb. am 18.07.2003
Staatsangehörigkeit:	Deutsch
Anschrift:	Beethovenstr. 38 D-86438 Kissing

Hochschulabschluss

Oct 1997: Diplom-Physiker, Technische Universität München
Thema: "Generation and Modification of Triangle Meshes from Trinocular Depth Maps",
Gutachter: Prof. Dr. Radig (Fakultät für Informatik, Technische Universität München)
Zweitgutachter: Prof. Dr. Ring (Physik-Department, Technische Universität München)

Beruflicher Werdegang

Nov 1997 - Apr 2000: Softing Ag, Haar bei München, Hardware- und Software-Entwicklung für industrielle Steuerungen

Mai 2000 - Jun 2006: Siemens Ag, München, Mobile Devices, Grundlagenforschung und Vor-Entwicklung

seit Jul 2006: MTU Aeoro Engines, München, Entwicklung von Regelungssystemen für Flugzeugtriebwerke

Hochschulbildung

Dez 1996 - Aug 1997: Diplomarbeit, Siemens Ag, Zentralabteilung Forschung und Entwicklung, München-Neuperlach im Rahmen des von der Europäischen Union geförderten Projektes "Panorama"

Apr 1993 - Okt 1996: Hauptstudium Physik, Nebenfach Informatik, Technische Universität München

Okt 1990 - Mär 1993: Grundstudium Physik, Nebenfach Informatik, Universität Augsburg

Forschungsinteressen

Compilerbau

Hardwarecompilierung

VLSI design

Parallele Programmierung